DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

ENERGY EFFICIENT ALGORITHMS

by

Canan BEŞEL

February, 2011

İZMİR

ENERGY EFFICIENT ALGORITHMS

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering, Computer Engineering Program

by

Canan BEŞEL

February, 2011

İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**ENERGY EFFICIENT ALGORITHMS**" completed by **CANAN BEŞEL** under supervision of **ASST. PROF. DR. GÖKHAN DALKILIÇ** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Gökhan DALKILIÇ

Supervisor

(Jury Member) (Jury Member)

Prof. Dr. Mustafa SABUNCU

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

During this research my supervisor supported me in every phase and guided me about the way of developing. Thanks to him for his support and for giving constructive suggestions to improve the quality of this thesis. Also with the encouragement of him I wrote an article on this subject and sent it to a conference named ISC Turkey. It has been accepted and I have made a representation within the concept of it.

Canan BEŞEL

ENERGY EFFICIENT ALGORITHMS

ABSTRACT

Nowadays, a variety of systems are used which have power supply constraints. It is important that all design efforts are made to conserve power in those systems. Energy consumption in a system can be reduced with hardware changes but application software running on the system has a key role in energy consumption too. In this thesis, the impact of various software implementation techniques on performance and energy saving is studied. It looks for strategies and types to decrease the execution time and the energy consumed by a given processor core when executing a program especially written in the C# language running on given input.

Keywords: Energy efficient algorithms, energy consumption of programs, code optimization.

ENERJİ TASAARUFLU ALGORİTMALAR

ÖZ

Günümüzde güç kısıtları olan çeşitli sistemler kullanılmaktadır. Bu sistemlerde gücü korumak için yapılabilecek çalışmalar önemlidir. Bir sistemdeki enerji tüketimi çeşitli donanım değişiklikleriyle azaltılabilir ancak sistemler üzerinde çalışan uygulama yazılımları da enerji tüketiminde önemli bir rol oynar. Bu çalışmada, performans ve enerji tasarrufu için uygulanabilecek çeşitli yazılım teknikleri üzerine çalışıldı. Çalışma içersinde özellikle C# dili kullanılarak yazılmış bir programın belirli bir girdiyle çalıştırılması sırasında işlemcinin çalışma zamanını ve enerji tüketimini azaltmak için kullanılabilecek strateji ve tipler araştırılır.

Anahtar sözcükler: Enerji tasarruflu algoritmalar, programların enerji tüketimleri, kod optimizasyonu.

CONTENTS

Page

M.Sc THESIS EXAMINATION RESULT FORM ii
ACKNOWLEDGMENTS iii
ABSTRACT iv
ÖZ v

CHAPTER ONE - INTRODUCTION 1

CHAPTER TWO - BACKGROUND 3

2.1 Instruction Level Optimization 3
2.1.1 Instruction Level Power Analysis 3
2.1.2 Computation of Energy Cost 4
2.1.3 Instruction Packing 8
2.1.4 Instruction Reordering 8
2.1.5 Reduction of Memory Operands 9
2.1.6 Operand Swapping in Booth Multiplier 9
2.1.7 Register Pipelining 10
2.2 Data Optimization 11
2.3 Algorithmic Optimization (Application Layer Optimization) 12
2.3.1 Object Oriented Programming Strategies 12
2.3.2 Avoid Polling 14
2.3.3 Multithreading 15
2.3.4 Reduce Usage of High-Resolution Periodic Timers 16
2.3.5 Loops 17
2.4 Other Optimization Techniques 17
2.5 Optimization in Mobile Application 18
2.5.1 Reads & Writes 20
2.5.2 Heap Usage 20
2.5.3 I/O Operations 21

CHAPTER THREE - PERFORMANCE TOOLS 22

3.1 Perfmon..22
3.2 Intel® Vtune™ Analyzer..23
3.3 CLR Profiler..23
3.4 SOS 24
3.5 VSTS Profiler...25
3.6 Windows Event Viewer/Event Log (Windows* XP & Windows Vista*)...25
3.7 Windows ETW (Windows* XP & Windows Vista*) 25
3.8 PowerInformer (Windows* XP & Windows Vista*) 25
3.9 PowerTOP (Linux)...26
3.10 Battery Life Toolkit (BLTK) (Linux) 26

CHAPTER FOUR - C# CODE OPTIMIZATION..27

4.1 Class vs. Struct 29
4.2 Static vs. Dynamic Variable 30
4.3 Recursion vs. Iteration 31
4.4 Function Usage...32
4.5 Parameter Order 34
4.6 ArrayList vs. Array 35
4.7 Foreach vs. For...36
4.8 String.Format vs. String Builder vs. Concatenation 37
4.9 Boxing-Unboxing...38
4.10 Reading Values of Objects Once 39
4.11 Special Operators 39
4.12 Parallel Programming 40
4.13 Smart Try-Catch - Minimize Exceptions..42

CHAPTER FIVE - DEVELOPMENT & TEST RESULTS 46

5.1 Dual Core Machine – 2 threads...50
5.1.1 50 words..51
5.1.2 100 words..52
5.1.3 150 words..52
5.1.4 300 words..52

5.1.5 600 words 53
5.1.6 1000 words 53
5.1.7 5000 words 54
5.1.8 10000 words 54
5.2 Two Quad Core Machine 55
5.2.1 50 words 56
5.2.2 100 words 56
5.2.3 150 words 57
5.2.4 300 words 57
5.2.5 600 words 58
5.2.6 1000 words 58
5.2.7 5000 words 58
5.2.8 10000 words 59
5.3 i7 - 8 Threaded Machine 59
5.3.1 50 words 60
5.3.2 100 words 60
5.3.3 150 words 61
5.3.4 300 words 61
5.3.5 600 words 62
5.3.6 1000 words 62
5.3.7 5000 words 62
5.3.8 10000 words 63
5.4 Results of tests 64
5.4.1 Performance results 64
5.4.2 Battery status results 66
5.4.3 Energy results 68
5.4.4 Summary of results 72

CHAPTER SIX 76

CONCLUSIONS 76

REFERENCES .... Hata! Yer işareti tanımlanmamış.

APPENDICES 82

CHAPTER ONE

INTRODUCTION

Since most day to day operations are moving online (reservations, core banking, shopping) and popularity of power constrained computers is increasing (notebooks, digital cameras, mobile phones), software performance has become vital to their success in terms of response time in web sites; battery life and heat dissipation for portable devices. For example, so many times visits to a web site take long time to load which can result with frustration and the migration to a different site. For businesses this can be fatal as they lose customers. For another example, an application designed for mobile phones can be useless as it consumes much power and causes a shorter battery life. Energy consumption is very crucial especially in terms of battery lives of portable devices.

Energy consumption in a system can be reduced with many technical improvements concerning the architecture of electronic systems. Most of them are from the area of hardware design. But beside changes in hardware design, software design issues are another promising approach (Steinke, Schwarz, Wehmeyer & Marwedel, 2001). Software can play an important role in reducing the power and extending the battery time. Furthermore, software changes are generally less expensive and can be delivered as an update (Steinke, Schwarz, Wehmeyer & Marwedel, 2001).

Every activity carried out by applications can affect the power consumption of any computer which can be defined as the energy consumption of software running on them. Software developers frequently face the problem of estimating how much energy and time are spent in their software. This is crucial to determine how fast their software runs on a platform, how much energy it consumes, where optimizations are needed, or what hardware it requires to ensure a given speed. And this problem is not effectively solved by current approaches like instruction-level simulation, static timing analysis and source-level instrumentation.

In light of the above, there is a clear need for considering the power consumption on systems from the point of higher levels of software and optimizing the source code for less power consumption. As the trend of applications goes in “object oriented programming”, this thesis tries to find the best ways, strategies, and types that can be used during software development with C# language for less energy consumption. The details in this study are as follows: There will be the mention about other studies related to low power consumption of software from the point of different levels as instruction, data and application level in Section 2. Next, the tools that can be used in observations about resource usage of software in different environments are listed in Section 3. Then in Section 4, the major characteristics of OOP those can substitutes to one another are raised with the results of resource consumption comparisons. In Section 5; some of the characteristics, types and strategies suggested in Section 4 for lower energy are used in an application together and the results are analyzed for performance and energy gain. Finally, some concluding remarks have been given.

CHAPTER TWO

BACKGROUND

The design of system software, the actual application source code and the process of code translation to machine instructions – all of these determine the power cost of the software. So optimizations can be applied at three levels of abstraction: instruction-level, data-level and application level. Several researches about energy efficient codes and optimization of codes to make them more efficient have been studied in the past. In this section, previous works about instruction level and data level optimization are analyzed in Section 2.1 and Section 2.2 consecutively. Then algorithmic level optimization studies and some other techniques are given in Section 2.3 and Section 2.4.

2.1 Instruction Level Optimization

In order to analyze and quantify power cost of the program, it is important to start from the most fundamental level. This is the level of individual instructions executing on the processor. Instruction level optimization is the optimization of software that translates a high-level language into machine code for the target microprocessor. It analyzes power consumption from the point of view of instructions. It provides too low-level information and it is too slow (Scarpazza, 2006).

2.1.1 Instruction Level Power Analysis

“Instruction level power analysis”, which is first proposed by Tiwari, is the technique used to provide the fundamental information needed to evaluate the power cost of the program (Tiwari, Malik & Wolfe, 1996). This technique estimates the energy cost of a program by summing the energy consumption of each instruction. Instruction-by-instruction energy costs are determined for each target processor which is called “base cost”. The base cost of an instruction is defined as the average current drawn by the processor when it is executed and it is measured with a program containing a loop of the individual instruction. Also there is another effect that impacts the overall power cost of programs. The “overhead cost” is the measure of circuit state change for a sequence of two different instructions. It is measured with the difference in the current of an infinite loop of a pair of different instructions with the average of the base costs of the instructions.

So the total energy consumed by a program P is given by Equation 1 (Tiwari, Malik & Wolfe, 1996).

$$E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \qquad (1)$$

'B_i' is the base cost of each instruction '*i*'.

'N_i' is used for the number of times the instruction '*i*' is executed.

'$O_{i,j}$' is the circuit state overhead when instruction '*i*' and instruction '*j*' are adjacent.

'E_k' is the energy overhead of the other inter instruction effects (stalls and cache misses).

2.1.2 Computation of Energy Cost

Table 2.2 shows CPU base costs for some Intel 486Dx2 processor instructions (Tiwari, Malik & Wolfe, 1996).

As an example consider a program containing a sequence of instructions like shown in Table 2.1.

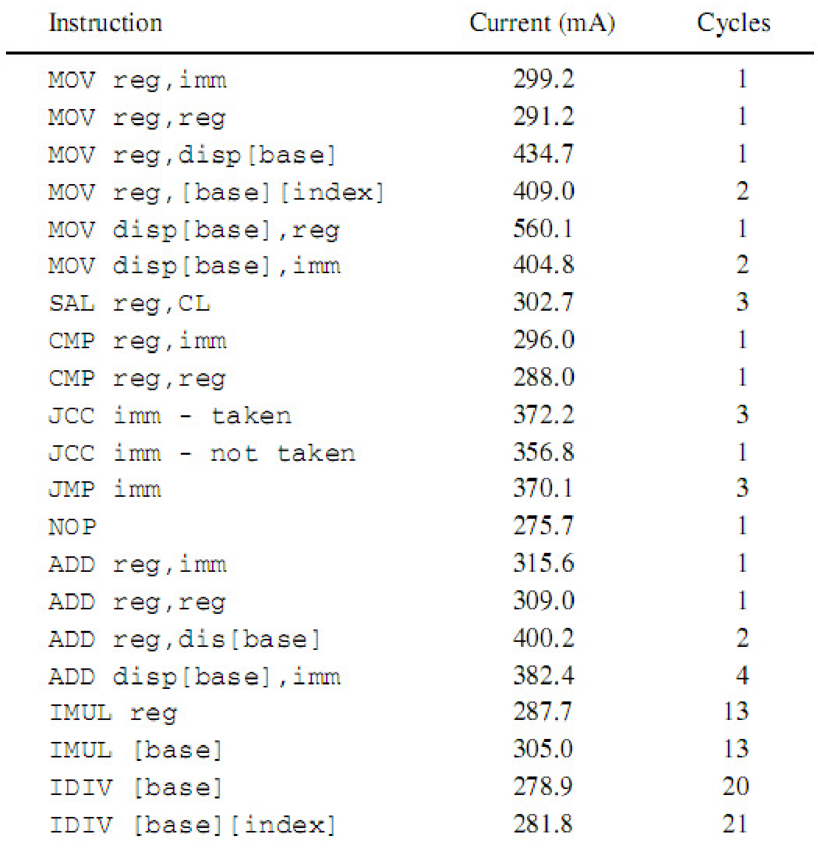
Table 2.1 An example instruction sequence

Number	Instruction
1	MOV CX,1
2	ADD AX,BX
3	ADD DX,8[BX]
4	MOV AX,BX
5	SAL BX,CL

The total cost of this sequence can be calculated using base costs like below:

MOV CX,1	➔ MOV reg,imm	➔ 299.2 mA * 1 cycle
ADD AX,BX	➔ ADD reg,reg	➔ 309.0 mA * 1 cycle
ADD DX,8[BX]	➔ ADD reg,dis[base]	➔ 400.2 mA * 2 cycles
MOV AX,BX	➔ MOV reg,reg	➔ 291.2 mA * 1 cycle
SAL BX,CL	➔ SAL reg,CL	➔ 302.7 mA * 3 cycles

Table 2.2 Subset of the base cost table for Intel 486Dx2



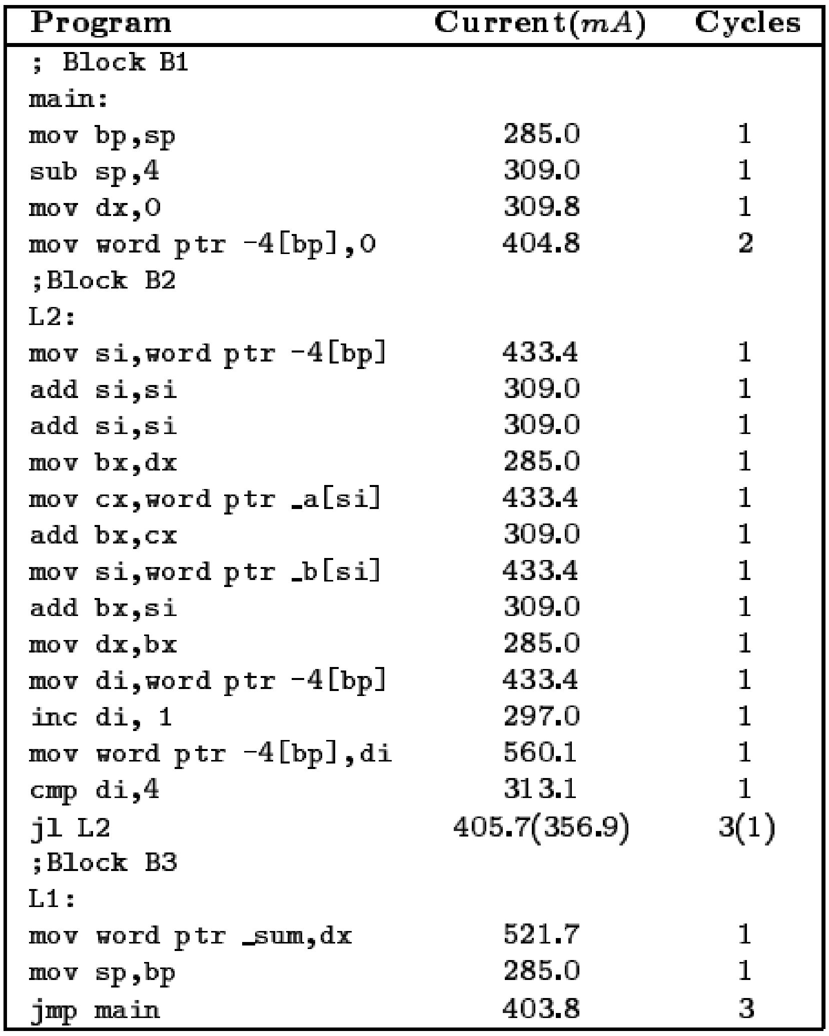
Instruction	Current (mA)	Cycles
MOV reg,imm	299.2	1
MOV reg,reg	291.2	1
MOV reg,disp[base]	434.7	1
MOV reg,[base][index]	409.0	2
MOV disp[base],reg	560.1	1
MOV disp[base],imm	404.8	2
SAL reg,CL	302.7	3
CMP reg,imm	296.0	1
CMP reg,reg	288.0	1
JCC imm - taken	372.2	3
JCC imm - not taken	356.8	1
JMP imm	370.1	3
NOP	275.7	1
ADD reg,imm	315.6	1
ADD reg,reg	309.0	1
ADD reg,dis[base]	400.2	2
ADD disp[base],imm	382.4	4
IMUL reg	287.7	13
IMUL [base]	305.0	13
IDIV [base]	278.9	20
IDIV [base][index]	281.8	21

To get a closer estimate we consider the circuit state overhead between each pair of consecutive instructions is known. The overhead values between the pairs 1&2, 2&3, 3&4, 4&5 and 5&1 are found to be 17.9 mA, 5.25 mA, 16.8 mA, 17.4 mA, 17.2 mA consecutively. So the total energy cost can be calculated by using Equation 1:

((299.2*1 + 309.0*1 + 400.2*2 + 291.2*1 + 302.7*3) + (17.9 + 5.25 + 16.8 + 17.4 + 17.2)) / 8 = 335.3 mA current over 8 cycles

To make this calculation on a program code, the code is converted to its equivalent assembly code containing instructions divided into blocks. Then the instructions' base costs and number of cycles of them are determined. For each block, base costs of the instructions are multiplied with the cycle number and the products are summed up to find the base energy cost of the block.

Table 2.3 An example instruction sequence



Program	Current(mA)	Cycles
; Block B1		
main:		
mov bp,sp	285.0	1
sub sp,4	309.0	1
mov dx,0	309.8	1
mov word ptr -4[bp],0	404.8	2
;Block B2		
L2:		
mov si,word ptr -4[bp]	433.4	1
add si,si	309.0	1
add si,si	309.0	1
mov bx,dx	285.0	1
mov cx,word ptr _a[si]	433.4	1
add bx,cx	309.0	1
mov si,word ptr _b[si]	433.4	1
add bx,si	309.0	1
mov dx,bx	285.0	1
mov di,word ptr -4[bp]	433.4	1
inc di, 1	297.0	1
mov word ptr -4[bp],di	560.1	1
cmp di,4	313.1	1
jl L2	405.7(356.9)	3(1)
;Block B3		
L1:		
mov word ptr _sum,dx	521.7	1
mov sp,bp	285.0	1
jmp main	403.8	3

As you see in the code, Block-2 in Table 2.3 is executed according to a condition. So cost of the *'jl L2'* statement is different according to whether the jump is taken or not. By multiplying the cost of the blocks with the number of times it is executed, adding the cost of the unconditional jump *'jl L2'* statement to it, dividing the result by the number of cycles, and at the end by adding the average circuit state overhead value to the result, the total energy cost can be calculated approximately. It can be summarized as:

```
While (!exit)
{
int i = 1;
double energy = 0;
begin
    energy += BLOCK1;
    while (i < 4){
         energy += BLOCK2;
         i = 1 + 1;
    if(jump is taken)
    {
       energy += cost_of_(jl L2);
    }
    }
energy += BLOCK3;
     end;
}
```

Modern compilers can make some optimizations automatically on the code that had been written by the programmers to make it run more efficiently. These compilers are called “optimizing compilers” (Aslan, 2006). The basic condition for the optimization is ‘not to upset the equivalence of the original code’, which means it should not change the meaning of the code. Figure 2.1 summarizes the compilation process of optimizing compilers.

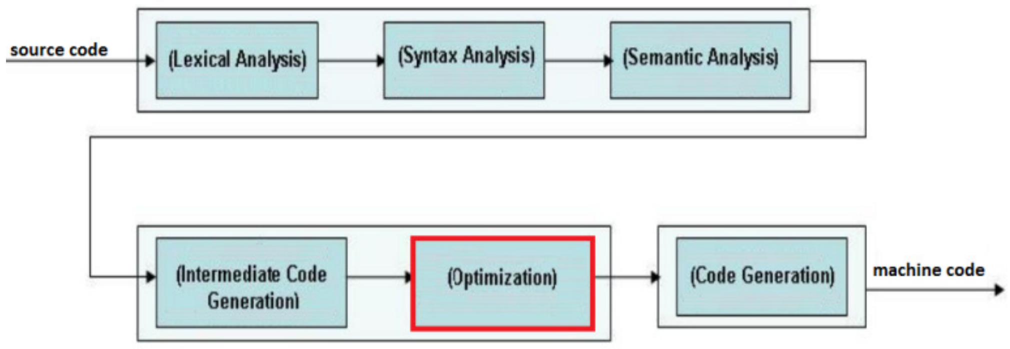


Figure 2.1 Compilation process of optimizing compilers

Techniques of instruction level optimization driven by the compilers are:

2.1.3 Instruction Packing

The DSP has a special architectural feature called instruction packing. It is the feature of packing an ALU-type instruction and a data transfer instruction into a single instruction codeword for a simultaneous execution. When the instructions are packed, it executes in one cycle and the circuit-state overhead current between two adjacent unpacked instructions is eliminated (Lee, Tiwari, Malik & Fujitsu, 1995).

Average current for the packed instructions is a bit more than unpacked instruction sequence but the unpacked instructions complete in twice the number of cycles as the packed instructions which results with a larger energy consumption as shown in Figure 2.2.

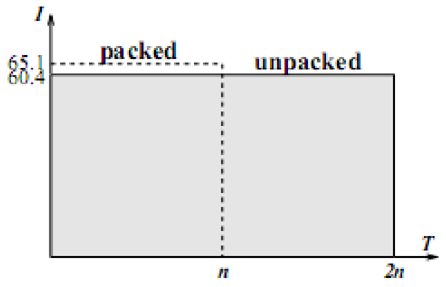


Figure 2.2 Comparison of energy consumption of packed and unpacked instructions

2.1.4 Instruction Reordering

The energy consumed during execution of an instruction depends on the previous instruction because of the switching activity in the circuit. Thus, order of the instructions affects the energy consumption of our programs. This means reordering the instructions can reduce the circuit-state overhead and minimize the energy consumption.

It has been observed that this technique lead to very little impact in the case of the 486DX2 and the ‘934 processors. But in the case of DSP, this impact is more significant (Tiwari, Malik & Wolfe, 1996). Effect of instruction reordering in the ‘934 can be seen in Table 2.4.

Table 2.4 Effect of instruction reordering in the ‘934

No.	Instruction	Register contents
1	fmuls %f8,%f4,%f0	%f8=0, %f4=0)
2	andcc %g1,0xaaa,%l0	(%g1=0x555)
3	faddd %f10,%f12,%f14	(%f10=0x123456, %f12=0xaaaaaa)
4	ld [0x555],%o5	
5	sll %o4,0x7,%o6	(%o4=0x707)
6	sub %i3,%i4,%i5	(%i3=0x7f, %i4=0x44)
7	or %g0,0xff,%l0	

	Sequence	Current (mA)
a	1,2,3,4,5,6,7	227.5
b	1,3,5,7,2,4,6	224
c	1,4,7,2,5,3,6	226
d	2,3,7,6,1,5,4	228
e	5,3,1,4,6,7,2	223.5

2.1.5 Reduction of Memory Operands

Instructions with memory operands have very high energy costs compared to instructions with register operands. Because register operands lead to shorter running times due to elimination of potential stalls and cache misses. Thus reduction in the number of memory operands can supply large energy savings (Tiwari, Malik & Wolfe, 1994). Reducing memory operands can be done with optimal register allocation of temporaries and global register allocation of most frequently used variables.

2.1.6 Operand Swapping in Booth Multiplier

The Booth multiplier implemented in the MAC unit takes the data in registers A and B as operands for multiplication as shown in Figure 2.3. But it does not treat A and B in the same way. B is recorded by a so-called “skipping over 1s” technique and A is added or subtracted for the number of times determined by B while executing the production process (Lee, Tiwari, Malik & Fujitsu, 1995).

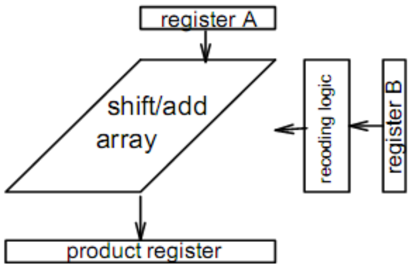


Figure 2.3 Microarchitecture model for the Booth multiplier

So if the weight of A is smaller than that of B, the number of addition and subtraction operations decreases and it supplies a reduction in current. As a result with just swapping the operands in a product instruction, current and power consumption can be reduced as shown in Table 2.5.

Table 2.5 Effect of operand swapping in power reduction

operands		measured current		
op1	op2	op1 * op2	op2 * op1	%saving
7FFFFF 000001	AAAAAA AAAAAA	58.9	46.9	20.4%
7FFFFF 000001	666666 AAAAAA	68.5	47.9	30.1%
7FFFFF 000001	AAAAAA 000001	65.7	49.1	25.3%

2.1.7 Register Pipelining

Arrays are usually stored in memory and the elements of them are accessed with load and store instructions. Register pipelining is a known optimization technique in compilers which eliminates these accesses in loops by temporarily storing the data in unused processor registers whenever this is possible (Steinke, Schwarz, Wehmeyer & Marwedel, 2001).

The main principle can be shown in C# code given below.

Original Code:

```
for (i = 1; i < 120; i++) {
                a[i] = a[i-1] + 3;
}
```

Optimized Code:
```
R = a[0];
   for (i = 1; i < 120; i++) {
      R = R + 3;
                 a[i] = R;
 }
```

2.2 Data Optimization

Code can be optimized by changing the representation of data manipulated by the algorithms to match the characteristics of the target architecture with the processed data (Simunic, Benini, Micheli & Hans, 1999).

Most processors execute faster if certain data values are aligned on word, double-word or page boundaries. So if possible, structures must be designed to satisfy appropriate alignments to avoid exceptions.

In an assembly language, the choice of a particular instruction or data type can have a large impact on execution efficiency. In general, instructions that process variables such as signed or unsigned 16-bit or 32-bit integers are faster than instructions that process floating point or packed decimal. Modern processors are even capable of executing multiple 'fixed point' instructions in parallel with the simultaneous execution of a floating point instruction. If the largest integer to be encountered can be accommodated by the 'faster' data type, defining the variables as that type will result in faster execution. Assembler programmers and optimizing compiler writers can then also benefit from the ability to perform certain common types of arithmetic (performing faster binary shift right operations instead of division).

If the choice of input data type is not under the control of the programmer, although prior conversion (outside of a loop for instance) to a faster data type carries some overhead, it can often be worthwhile if the variable is then to be used as a loop counter, especially if the count could be quite a high value or there are many input values to process. As mentioned above, choice of individual assembler instructions (or even sometimes just their order of execution) on particular machines can affect the efficiency of an algorithm. Sometimes microcode or hardware quirks can result in unexpected performance differences between processors that assembler programmers can actively code for something even the best optimizing compiler may not be designed to handle.

2.3 Algorithmic Optimization (Application Layer Optimization)

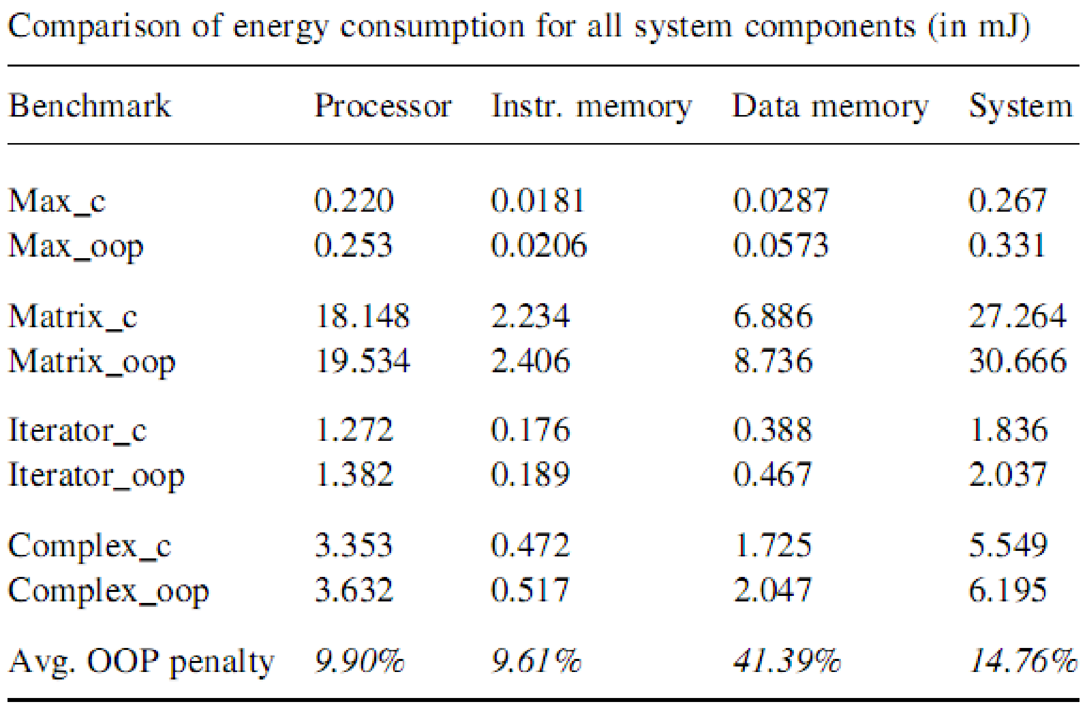
Highest layer in the optimization hierarchy targets algorithms. The choice of the algorithm and other high level decisions about the design of the software can affect the energy consumption.

This layer has the most information on the actual user impact of performance and energy tradeoffs. Application-specific optimizations can be made at this layer such as changing the algorithm used, accuracy of computation (eg. changing from double precision to single), or quality of service provided. For a particular problem, a stack may be better than a queue and a B-tree may be better than a binary tree or a hash function. The best algorithm or data structure to use depends on many factors, which indicates that a study of the problem and a careful consideration of the architecture, design, algorithms, and data structures can lead to an application that performs better and consumes less energy. Also, energy usage at the application layer may be made dynamic. For instance, an application hosted in a data center may decide to turn off certain low utility features if the energy budget is being exceeded, and an application on a mobile device may reduce its display quality when battery is low.

There are several ways and techniques that can be made in application layer. Previous works done in the concept of this layer are given shortly in this section.

2.3.1 Object Oriented Programming Strategies

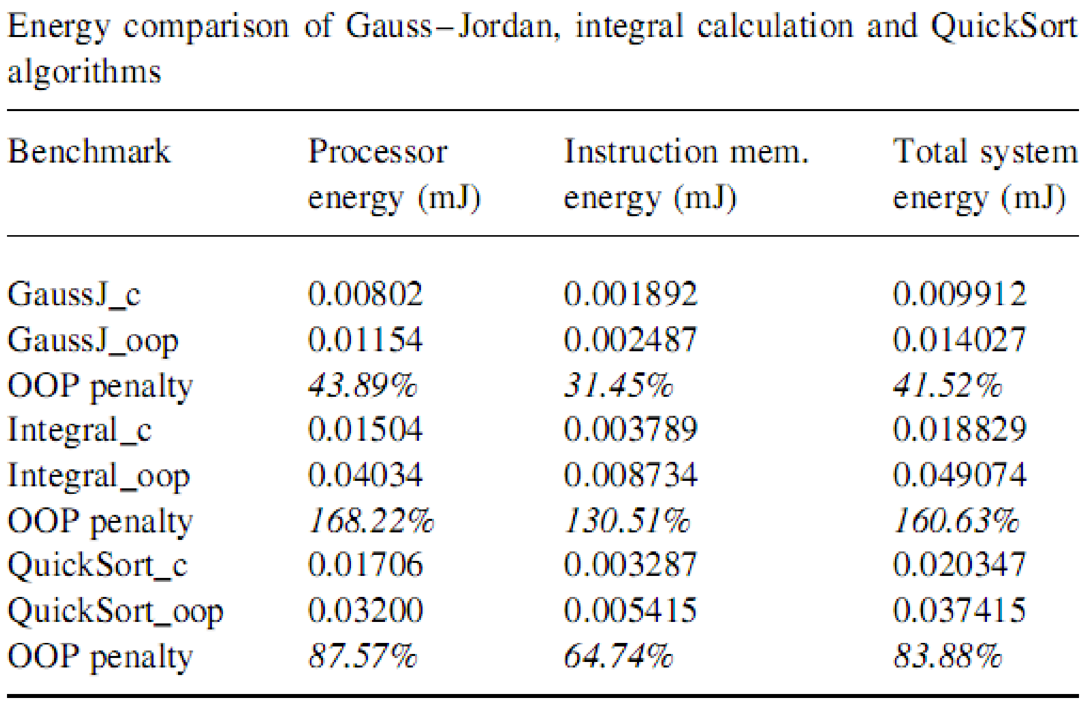
Chatzigeorgiou (2002) emphasizes on that the object-oriented approach shows a significant performance penalty compared to classical procedural programming due to the increased instruction count, larger code size and increased number of accesses to the data memory. According to this study, energy consumption penalty of object oriented programming compared to classical procedural programming (C vs. C++) can be seen in Figure 2.4 and Figure 2.5 (Chatzigeorgiou, 2002).



Comparison of energy consumption for all system components (in mJ)

Benchmark	Processor	Instr. memory	Data memory	System
Max_c	0.220	0.0181	0.0287	0.267
Max_oop	0.253	0.0206	0.0573	0.331
Matrix_c	18.148	2.234	6.886	27.264
Matrix_oop	19.534	2.406	8.736	30.666
Iterator_c	1.272	0.176	0.388	1.836
Iterator_oop	1.382	0.189	0.467	2.037
Complex_c	3.353	0.472	1.725	5.549
Complex_oop	3.632	0.517	2.047	6.195
Avg. OOP penalty	*9.90%*	*9.61%*	*41.39%*	*14.76%*

Figure 2.4 Comparison of energy consumption



Energy comparison of Gauss–Jordan, integral calculation and QuickSort algorithms

Benchmark	Processor energy (mJ)	Instruction mem. energy (mJ)	Total system energy (mJ)
GaussJ_c	0.00802	0.001892	0.009912
GaussJ_oop	0.01154	0.002487	0.014027
OOP penalty	*43.89%*	*31.45%*	*41.52%*
Integral_c	0.01504	0.003789	0.018829
Integral_oop	0.04034	0.008734	0.049074
OOP penalty	*168.22%*	*130.51%*	*160.63%*
QuickSort_c	0.01706	0.003287	0.020347
QuickSort_oop	0.03200	0.005415	0.037415
OOP penalty	*87.57%*	*64.74%*	*83.88%*

Figure 2.5 Comparison of energy consumption

Though it is known that OOP has quite much more overhead than assembly and procedural languages, development trend still heads to this new world. There are optimized strategies in writing OOP software under energy concerned environment.

According to the study done in 2006 by Chantarasathaporn and Srisa-an, there are some major characteristics and significant usages of OOP those can substitutes to one another. The results of resource consumption comparisons among the comparable commands are as follows:

- Static variable consumes more power than the dynamic one because it takes around 40% longer time than dynamic.
- Interface is more restrictive since the methods inside must not have method body while Abstract Class can have some attributes or method bodies, just at least only one class is abstract. There is no significant different between using Abstract Class and Interface in similar situation.
- Dynamic variable works slower than the static around 40%.
- Dynamic method runs faster than the static around 50%. Anonymous dynamic method is very CPU intensive and it takes around 80% longer time than regular dynamic method.
- When using dynamic class attribute locally, users may just use it barely or use with "this" keyword. There is no significant difference in term of CPU usage of this pair.
- The most CPU consuming field is protected variable while private and public ones spend time quite close to each other. Protected attribute is slower than the other two around 40%.

2.3.2 Avoid Polling

Polling refers to actively sampling the status of an external device by a client program as a synchronous activity. Some examples of how applications perform unnecessary polling include (LessWatts,n.d.):

- Checking every second to see if the mouse moved
- Check every second to see if it is time to show the next minute on the clock

- Check 10x/sec to see if the smartcard reader got inserted on USB
- Check if new data is added to database that must be shown on the screen

In applications, periodic polling seems to have become an easy, simple solution for many application problems. Every time an application polls for something, the CPU wakes from idle state and wastes power (LessWatts, n.d.). So it must be avoided polling at all costs. Instead of this, event and notification architecture can be used. But sometimes it is really needed to use them so at these situations, polling interval can be increased. Polling not more often than one per second may be a better solution.

2.3.3 Multithreading

Execution can be speed-up by taking advantage of multiple threads. With multithreaded applications, the job may be able to finish in shorter time than single-threaded applications. Thanks to the increased idle time it supplies, it leads to energy savings as compared to a single-threaded version. But threads must be used correctly. If the threads are imbalanced it may lead to increased energy consumption (Steigerwald, Chabukswar, Krishnan & Vega, 2007).

In imbalanced threading there is a significant difference in the amount of work done by each thread within an application and the results indicate that the imbalanced threading model/under-utilized CPU may cause degradation in performance, causing increased power consumption.

In balanced threading each thread has an equal amount of work as other active threads of the application. Figure 2.6, Figure 2.7 and Figure 2.8 show performance, CPU power consumption and platform power consumption data for running single-threaded (ST) and multi-threaded (MT) versions of several CPU-intensive applications (Steigerwald, Chabukswar, Krishnan & Vega, 2007). The multithreaded applications clearly show significant performance improvements over running single-threaded versions. For example, the ST version of cryptography takes ~50 seconds to complete, while both the MT-1 and MT-2 versions take only ~25 seconds.

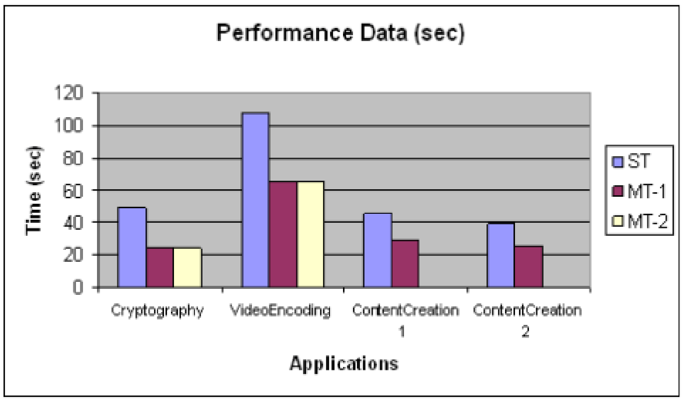


Figure 2.6 Balanced threading performance

Multithreading also saves power as shown in following figures. For example, the cryptography ST version running for ~50 seconds consumes ~150 mWHr of total power, while running the cryptography MT version for ~25 seconds and idling the system for the remaining 25 seconds consumes ~110 mWHr of total power.

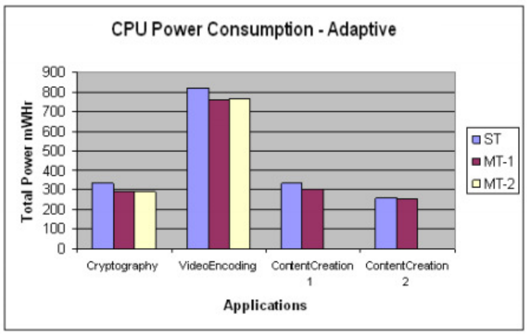


Figure 2.7 Balanced threading CPU power

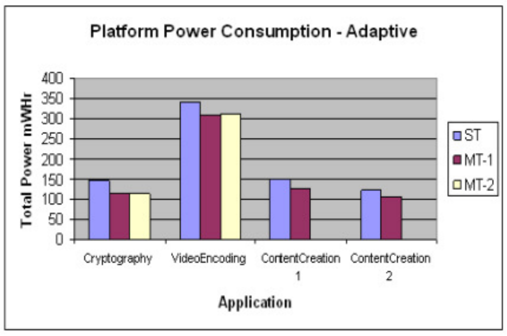


Figure 2.8 Balanced threading platform power

The results indicate that multithreading done correctly not only shows performance improvements but also saves power (Steigerwald, Chabukswar, Krishnan & Vega, 2007).

2.3.4 Reduce Usage of High-Resolution Periodic Timers

A good way of reducing energy is to let it idle as often as possible. Make sure the application is optimized to use the longest timer rate possible while fulfilling the requirements. Using timer intervals shorter than 15ms has small benefit for most applications. Always make sure to disable periodic timers in case they are not in use, letting the OS adjust the minimum timer resolution accordingly (Larsson, 2008).

2.3.5 Loops

Minimize the use of tight loops. To reduce the overhead implied with small loops, performance/power can be improved by performing loop unrolling. To achieve this, the instructions that are called in multiple iterations of the loop are combined into a single iteration. This will speed up the program if the overhead instructions of the loop impair performance significantly. Side effects may include increased register usage and expanded code size (Larsson, 2008).

2.4 Other Optimization Techniques

There are some other potential sources of energy reduction that can be applied during compilation whose effectiveness may be smaller as the methods described earlier. But any sources of energy reduction should not be ignored.

- Identify the kernel, drivers and libraries utilized by the application. Determine if there are alternative implementations of used components that are more power friendly. For instance, a more recent Linux kernel may feature scheduling optimizations making the application run more efficient. Another example would be to update to a more recent and energy efficient Bluetooth device driver (Larsson, 2008).
- If possible consider using a programming language implementation and libraries that are idle power friendly. Some high level run-time languages may cause more frequent wakeups compared to low(er) level system programming languages such as C (Larsson, 2008).
- Scheduling can be done to reduce pipeline stalls which takes up cycles and consume energy (Tiwari, Malik & Wolfe, 1994).
- Code transformations can be done to improve cache hit rates (Tiwari, Malik & Wolfe, 1994).
- Reducing switching in address lines (Tiwari, Malik & Wolfe, 1994).

- Improving page hit ratio. Because page misses in page-mode DRAM chips consume more energy (Tiwari, Malik & Wolfe, 1994).
- Don’t use too many Reflection API’s: Reflection API’s depend on the metadata embedded in assemblies. Thus parsing and searching this information is very expensive (Rodriguez & Dutta, 2008).
- Don’t make functions unnecessarily virtual or synchronized: JIT might disable some optimizations and so the generated code might not be optimal (Rodriguez & Dutta, 2008).
- Don’t write big functions: JIT might disable optimizations for faster compile (JIT) time (Rodriguez & Dutta, 2008).
- Choose the right framework for the scenario, including energy efficiency goals (Stemen, 2008).
- Try to use less complex (and more energy efficient) algorithms. For instance, select a lower quality video encoder/decoder when running on batteries (Stemen, 2008).
- Animations always increase system power consumption with extra CPU and memory utilization. So it must be avoided as possible (Stemen, 2008).

2.5 Optimization in Mobile Application

Usage of mobile applications and mobile computing has a growing popularity and energy is a vital resource for these systems as battery life and heat dissipation.

Everybody wants ‘all-day mobile pc battery life’. Users complain about short battery lives of their portable devices. So, extending battery life as long as possible is important, but how? You can see people saying ‘I have a notebook whose battery life is 8 hours’. But doing what; with playing DVD, with playing game or doing nothing? At this point impact of software comes out. There are studies in battery technology and low-power circuit design but studies in hardware scope cannot meet all the energy needs of future-mobile computers, improvements must be done in the higher levels of the system too. In other words software and energy consumption of them becomes more important in mobile systems.

Nowadays there are hundreds of different mobile models in mobile market which have all different characteristics including different systems. So at this point it becomes important to supply an application supported by much more devices. On the web there are two browsers and two or three operating systems that you have to support, if your application has been tested on them, you know that over 90 percent of your target audience will be able see and access your work. But in the mobile market, you deal with thousands of mobile devices with varying screen sizes and capabilities, operating systems and browsers. Content that looks great on one device may look odd or even unreadable on another.

How do you today ensure that your mobile content works consistently on the different devices? And how do you know what is "good" performance for your application? Performance in general means some characteristics that may be somehow measured. You can look at RAM usage, execution time, booting time, CPU usage and so forth. But in case of mobile applications, you have very limited resources available and there are strict requirements related to device characteristics and features. Therefore, mobile applications should be designed carefully and employ every possibility to improve their performance. While developing a mobile application, these can be done (Stemen, 2008):

- Firstly understand the impact of the software on platform power consumption.
- Focus on idle: how much energy it consumes in idle state, how can it be decreased, how can we get the system idle as long as possible.
- Reduce resource utilization: disk time, CPU time, memory alignment, sleep and resume transitions…
- Adapt to the system environment: what is the right tool for the job, what kind of application you should make and what kind of functionality it should have.
- Correctly handle sleep and resume transitions.

A good user experience and longer battery life are critical factors for the future growth of mobile systems. Software of applications running on these mobile systems has a key role to play in improving user experience as well as in extending battery life. Most of the optimization techniques listed in Section 2.3.1 can be applied in mobile applications too but there are subjects that are specific to mobile applications. Some of the points that mobile developers must care during development are listed next.

2.5.1 Reads & Writes

If a mobile application is moved to an upper version of the environment or if you work with some kind of flash card instead of the internal device's memory, operations with files can became dramatically slower. These are all because of read/write operations depend on the flash block size, regardless of how much data is read from or saved to the flash card. So, knowing this block size and adjusting buffers while developing applications accordingly can increase throughput of I/O operations (Gusev, 2006).

2.5.2 Heap Usage

On mobile devices, the stack size is often limited, so a heap should be used instead. But this also may cause performance to decrease when used unnecessarily (Gusev, 2006). Consider the following code:

```
while (expression)
{
    XXX *pObj = new XXX;
    DoSomething(pObj);
    delete pObj;
}
```

If this is a tight loop, many heap calls will cause heap fragmentation. In this case temporary variables must be used like the code below to increase performance:

```
  XXX *pObj = new XXX;
  while (expr)
  {
     DoSomething(pObj);
     pObj->Reset();
  }
 delete pObj;
```

2.5.3 I/O Operations

I/O operations have an important effect on performance in mobile applications. For desktop systems it is simple: read by blocks instead of bytes. But for mobile applications it is not as straightforward. If data is stored on a flash card then access time may be very long. Suppose that data is kept in a flat file as binary or text. It is a good thing if you can read it all in one time to memory and then process as needed. But in case of huge amounts of data, this is impossible. In those cases, you have to allocate chunks here and there. It is a really bad thing that memory allocation strategies may vary from one version of an OS to next one. On Pocket PC 2002 big allocations are good for performance, but on later versions smaller chunks are allocated faster. It is really hard to choose the best method to reach the best I/O performance (Gusev, 2006).

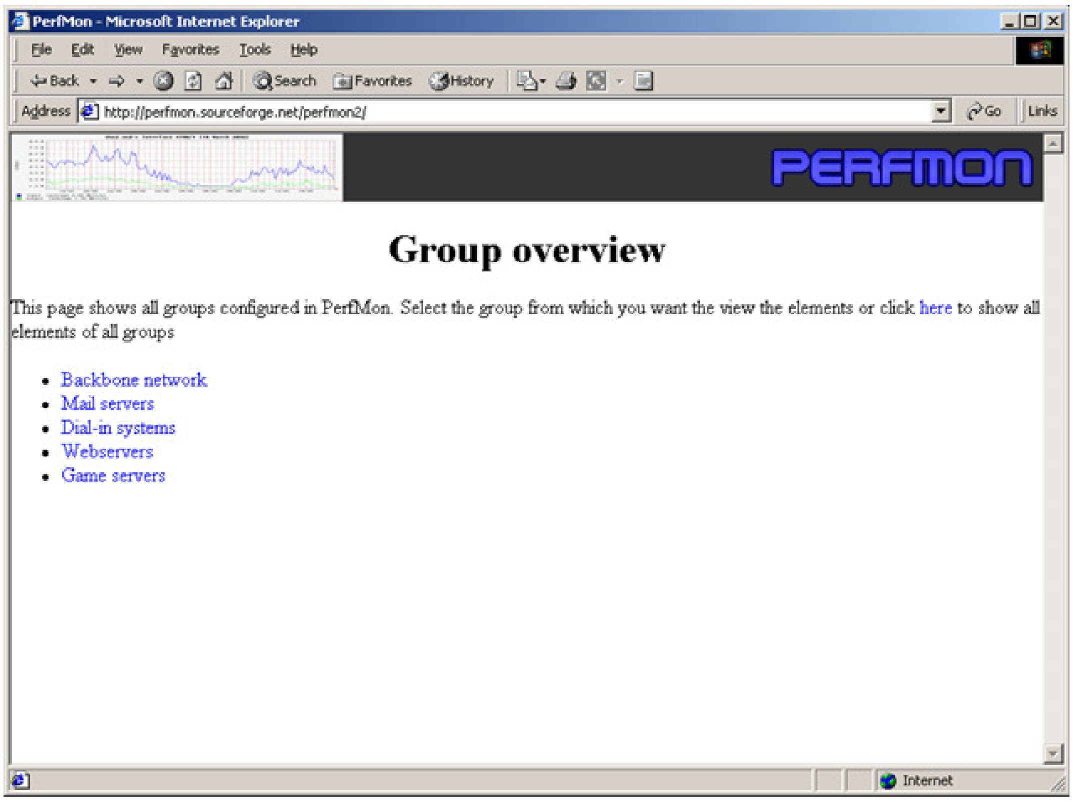
CHAPTER THREE

PERFORMANCE TOOLS

There are various tools that can be used in various systems for observing resource usage and performance of applications.

3.1 Perfmon

Perfmon is a system level tool that allows user-level code to access several ASP.NET related performance counters (Larsson, 2008). It can be used in analyzing any .Net, monitoring results of tuning and configuration scenarios, and the understanding of a workload and its effect on resource usage to identify bottlenecks. Some of example screenshots are shown in Figure 3.1 and Figure 3.2.


PerfMon - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://perfmon.sourceforge.net/perfmon2/
Go Links
PERFMON
Internet

Group overview

This page shows all groups configured in PerfMon. Select the group from which you want the view the elements or click here to show all elements of all groups

- Backbone network
- Mail servers
- Dial-in systems
- Webservers
- Game servers

Figure 3.1 Screenshot of Perfmon

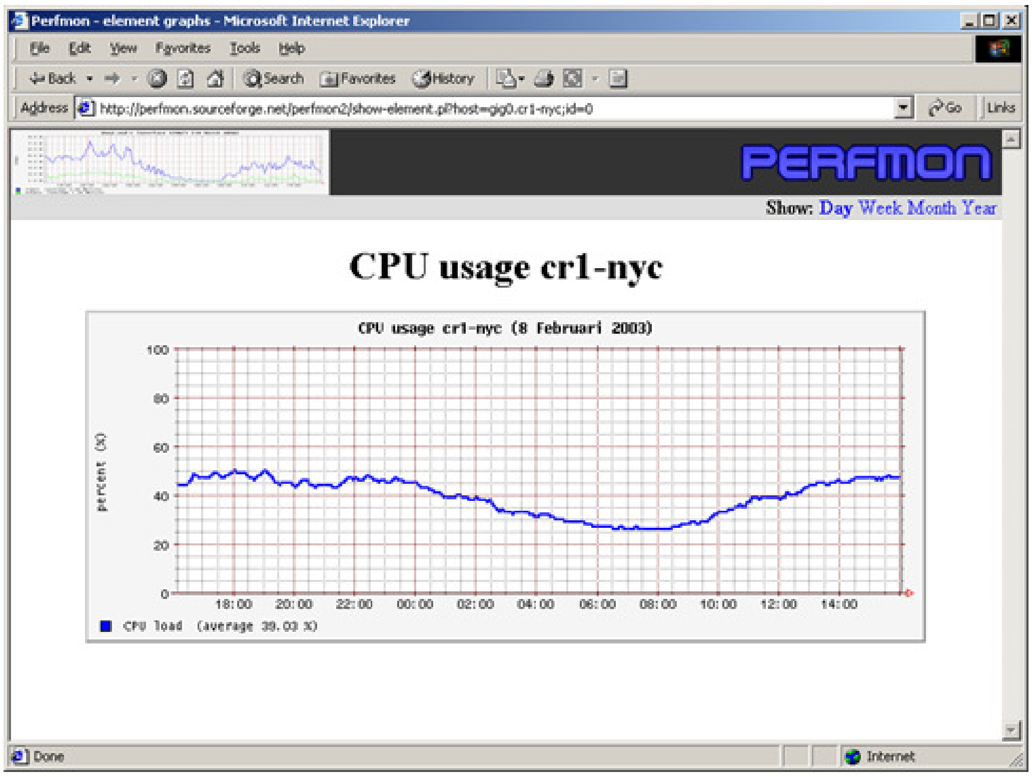

Perfmon - element graphs - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://perfmon.sourceforge.net/perfmon2/show-element.pl?host=gig0.cr1-nyc;id=0
PERFMON
Show: Day Week Month Year
CPU usage cr1-nyc
CPU usage cr1-nyc (8 Februari 2003)
percent (%)
100
80
60
40
20
0
18:00
20:00
22:00
00:00
02:00
04:00
06:00
08:00
10:00
12:00
14:00
CPU load (average 39.03 %)
Done
Internet

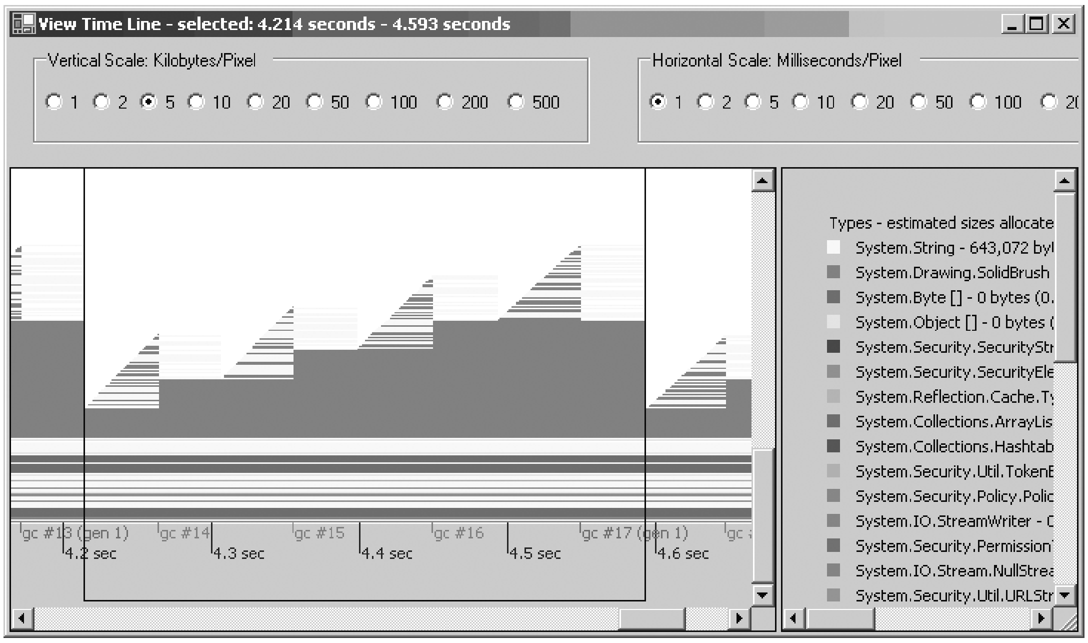
Figure 3.2 Screenshot of Perfmon

3.2 Intel® Vtune™ Analyzer

It is a profiling tool from Intel which supports .NET including ASP.Net applications (Larsson, 2008). It evaluates applications on all sizes of systems based on Intel processors to help improving application performance and makes application performance tuning easier.

3.3 CLR Profiler

It is a profiler tool from Microsoft which is used to profile memory allocation of applications and allows the user to investigate the contents of the manage heap as well as the behavior of the garbage collector, to identify portions of code which use too much memory. Some example screenshots are shown in Figure 3.3 and Figure 3.4 (Rodriguez & Dutta, 2008).

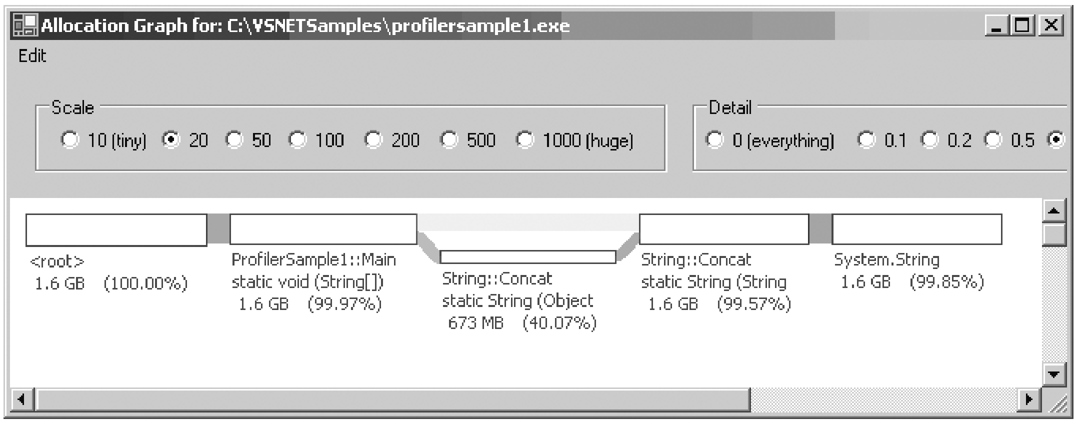

View Time Line - selected: 4.214 seconds - 4.593 seconds

Vertical Scale: Kilobytes/Pixel
1 2 [✓ 5] 10 20 50 100 200 500

Horizontal Scale: Milliseconds/Pixel
[✓ 1] 2 5 10 20 50 100 2

Types - estimated sizes allocate
- ▢ System.String - 643,072 by
- ■ System.Drawing.SolidBrush
- ■ System.Byte [] - 0 bytes (0.
- ▢ System.Object [] - 0 bytes (
- ■ System.Security.SecurityStr
- ■ System.Security.SecurityEle
- ■ System.Reflection.Cache.Ty
- ■ System.Collections.ArrayLis
- ■ System.Collections.Hashtab
- ■ System.Security.Util.TokenB
- ■ System.Security.Policy.Polic
- ■ System.IO.StreamWriter - C
- ■ System.Security.Permission
- ■ System.IO.Stream.NullStrea
- ■ System.Security.Util.URLStr

Figure 3.3 Screenshot of CLR Profiler


Allocation Graph for: C:\VSNETSamples\profilersample1.exe

Edit

Scale
10 (tiny) [✓ 20] 50 100 200 500 1000 (huge)

Detail
0 (everything) 0.1 0.2 0.5 [✓]

<root>
1.6 GB (100.00%)

ProfilerSample1::Main
static void (String[])
1.6 GB (99.97%)

String::Concat
static String (Object
673 MB (40.07%)

String::Concat
static String (String
1.6 GB (99.57%)

System.String
1.6 GB (99.85%)

Figure 3.4 Screenshot of CLR Profiler

3.4 SOS

It is the tool that exposes many CLR internal data structures such as GC, Exceptions, Objects, Locking etc. It can be used to identify functionality bugs (such as OutOfMemoryException) and performance related bugs as well (locking etc) (Rodriguez & Dutta, 2008).

3.5 VSTS Profiler

It is a built in profiler from Microsoft Visual Studio Team system 2008. It can be used in sampling application and identifying hotspots and hot call chains etc. It has ability to look at perfmon counters of all the machines from a client system, etc (Rodriguez & Dutta, 2008).

3.6 Windows Event Viewer/Event Log (Windows* XP & Windows Vista*)

It provides a centralized log service to report events that have taken place, such as a failure to start a component or to complete an action. For instance the tool can be used to capture “timer tick” change events which have an indirect effect on platform energy efficiency (Larsson, 2008).

3.7 Windows ETW (Windows* XP & Windows Vista*)

It provides application programmers the ability to start and stop event tracing sessions, instrumenting an application to provide trace events, and consume trace events. Events can be used to debug an application and perform capacity and performance analysis (Larsson, 2008).

3.8 PowerInformer (Windows* XP & Windows Vista*)

It provides relevant and condensed platform power information to the developer, including for instance battery status, interrupt rate and disk/file IO rates (Larsson, 2008).

3.9 PowerTOP (Linux)

It is a tool that can be used to point out the power inefficiencies of platforms. The tool shows how well the platform is using the various hardware power-saving features and culprit software components that are preventing optimal usage. It also provides tuning suggestions on how to achieve low power consumption (Larsson, 2008).

3.10 Battery Life Toolkit (BLTK) (Linux)

It provides infrastructure to measure laptop battery life, by launching typical single-user workloads for power performance measurement (Larsson, 2008).

CHAPTER FOUR

C# CODE OPTIMIZATION

Software optimization is generally done with speed and source usage aims. In other words, we work for faster applications or applications that need smaller memory. Of course it is willing to realize both of them but usually these two goals are coincided to each other. To speed up the code it is inevitable to enlarge it. Or shrinking the code can cause it to work slower. At this point, which one is more important? To speed up or to shrink the code? Speed of the code is dominant here. Generally, we have enough memory and speeding up helps our program more. For example imagine that you have to write a program aimed at the system and a function will be called for thousands time during the program. In this case a delay of 0.01 milliseconds will have very important effect on speed. Of course this situation can change in embedded systems where memory limited small microprocessors are used. So, the goal is to complete a task more quickly.

It is generally accepted that if the CPU can accomplish the task in fewer instructions or by doing work in parallel in multiple cores, and then drop the CPU to a low-power state, then the overall energy required to complete the task will be lower. Especially, current processors are quite good about saving power when idle, so making it to be idle longer will help to consume less energy. This behavior is called race-to-idle and can be explained with a simplified example:

Take a typical commercially available processor that consumes 34 Watts when running at full speed, and 24 Watts when running at half speed and 1 Watts when idle. On this processor, decoding one second of a MP3 file or some HDTV media every second takes 0.5 seconds at half speed, and, consequently, 0.25 seconds at full speed. The energy consumption for one second is:

Half speed: 0.5s * 24W + 0.5s * 1W = 12.5 Joules

Full speed: 0.25s * 34W + 0.75s * 1W = 9.25 Joules

As a result, it's generally better to run as fast as possible so that it can be idle longer which means less energy consumption.

In the past, both specific optimized equipments and codes were designed to relief this concern. This way worked in the past however, in this era, there is another significant restraint now, the time to market. To be able to prepare products in shorter period, object-oriented programming (OOP) has stepped in to this field. This new style heads to development methodologies, although it is known that it has quite much more overhead than assembly and procedural languages. It has been reported that OOP consume much resource (Chantarasathaporn & Srisa-an, 2006) which contradicts with the target of low power consumption, but it is accepted due to business reasons. Because of this, the language chosen for studying in this research is C#, based on .NET Framework 4.0 which is one of the trendy OOP development environments.

By the time your program is working, you might already know which functions and modules are the most critical for overall code efficiency. We can focus to those routines in which the program spends most (or too much) of its time. Once you've identified the routines that require greater code efficiency, you can use the following techniques to reduce their execution time.

The strategies and types that are compared in this research are tested with loops containing different code that's being tested for performance, with a time reading before and after. When the test has finished, the start time is subtracted from the end time to find the time cost. Usually the code run slower at the first execution, so several tests are done and the first 10 results are shown in x axis of graphs in this research. Also, the vertical axis points the total execution times (in milliseconds) of the tested code in different times of loops in each case. After strategies and their results, a list of words are encrypted and decrypted with AES in a tight loop and the results of the first 10 tests will be given for this data. Lastly, the test is done for different sizes of data. Then, near performance, energy consumptions of the original and optimized code are compared by using an example tight loop with battery status check before and after the test. It’s checked if one of the type or strategy being compared cause the battery to decrease more, especially to see if the energy consumption is related to execution time or not. As you will see in the test results too, at the end we can generally reach to the result that ‘the more timespan the process takes the more power the process spends’. The strategy used during this work can be seen in appendix.

Note that the techniques described here are very compiler-dependent. In most cases, there aren't general rules that can be applied in all situations. These options and strategies that had been compared here can be listed as:

- Class vs. Struct
- Static vs. Dynamic Variable
- Recursion vs. Iteration
- Function Usage
- Parameter Order
- ArrayList vs. Array
- Foreach vs. For
- String.Format vs. String Builder vs. Concatenation
- Boxing-Unboxing
- Reading Values of Objects Once
- Special Operators
- Parallel Programming
- Smart Try-Catch - Minimize Exceptions

4.1 Class vs. Struct

Firstly, the data-member-only classes and structs are compared. Both of them can contain group of variables or data members, but, as you see in Figure 4.1, it is easy to distinguish the difference of time spent.

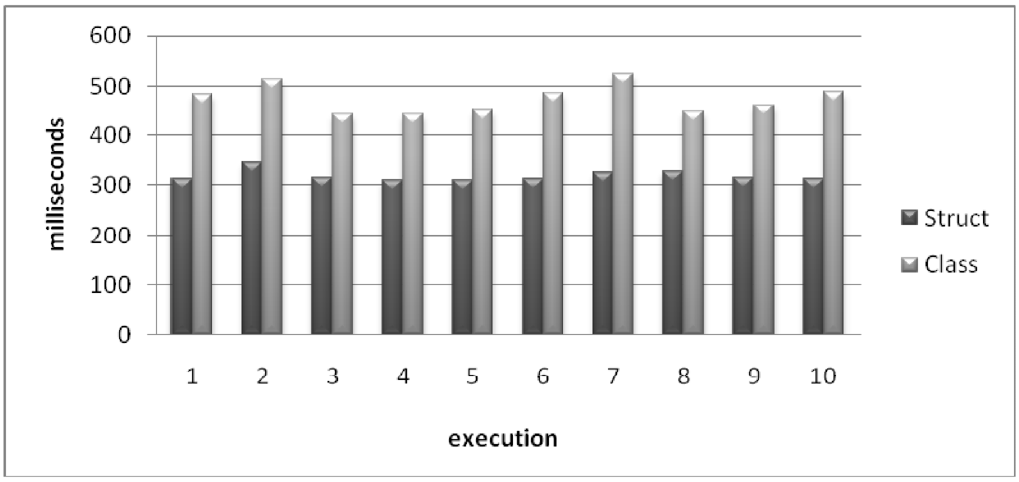


Figure 4.1 Class vs. Struct

In a tight loop the effect of this choice on the energy consumption can be seen. One of the test results are given below:

Using class: 13 minutes 12 seconds (%98 - %82)

Using struct: 10 minutes 34 seconds (%98 - %85)

4.2 Static vs. Dynamic Variable

Static variables are stored into RAM before the execution of code and they are hold in RAM during the program. So, these variables are not affected from the load and remove operations in the program. Thanks to the easiness of their address calculation, they are faster than dynamic variables as shown in Figure 4.2.

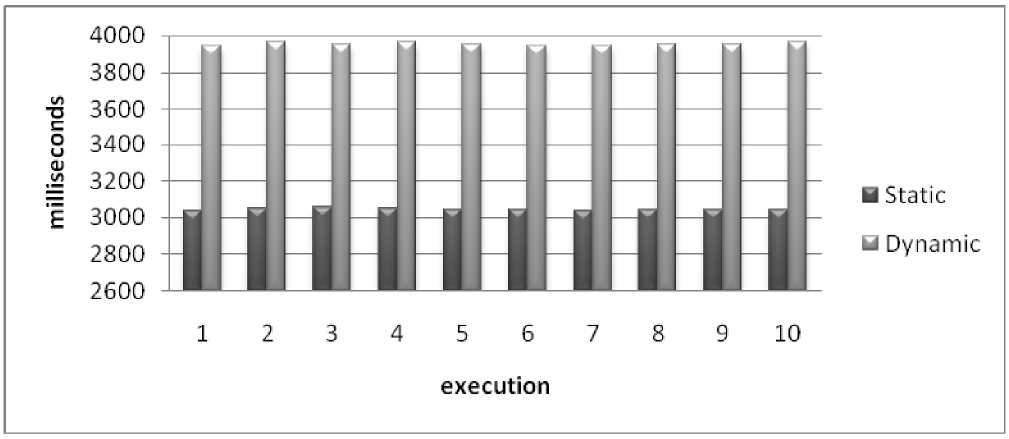


Figure 4.2 Static vs. Dynamic variable

Near its performance gain, using static variable instead of dynamic variable also saves power as can be seen from the test results given below:

Using dynamic variable: 22 minutes 28 seconds (%93 - %64)

Using static variable: 21 minutes (%93 - %67)

4.3 Recursion vs. Iteration

Recursion is a function that calls itself iteratively until it reaches a deadline. For some problems, designers can both use recursion or iteration. Recursive style is compact but sometimes it is more important to write faster code than writing more comprehensible code. This is why iteration is chosen most of the time. Due to the necessity of a stack to manage the recursion, it takes more time as shown in Figure 4.3. The results also show the differences on speed of two strategies.

Original source code:

```
private int TestRecursive(int p1)
    {
             if (p1 <= 1) return p1;
          int result = p1 + TestRecursive(p1 - 1);
          return result;
    }
```

Optimized source code:

```
 private int TestNonRecursive(int p1)
    {
          int result = 0;
          while (p1 > 0)
          {
             result = result + p1;
             p1--;
          }
          return result;
    }
```

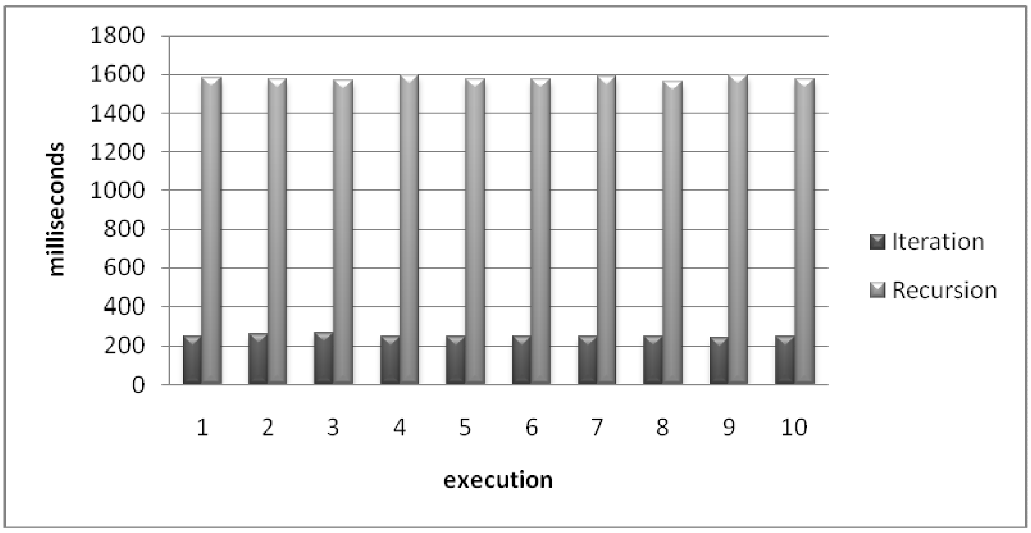


Figure 4.3 Recursion vs. Iteration

Using recursion instead of iteration increases memory usage of the code while causing it to run slower. Near these, energy consumption of the code increases too and it can be seen from the test results shown below:

Using recursion: 5 minutes 2 seconds (%99 - %94)

Using iteration: 2 minutes 34 seconds (%99 - %97)

4.4 Function Usage

Functions are basic building stones of structural programming. Functions have important impact on the size and speed of our code. When a compiler comes across with a function, it stores the parameters (if exist), output variables and the local variables that are used during the function in a stack. When the function is called, all these stored information is taken back from the stack (Yağmur, 2004). These operations take time, sometimes more than we imagine as can be seen in Figure 4.4. As a result, sometimes we should use local variables instead of these operations. But when? When the speed and time is important for our application. For example; if we have an application that does heavy mathematical operations. But of course while doing this, we should not to forget that, this will cause our application to enlarge.

Original source code:

```
 for (int x = 1; x < 10000000; ++x)
   {
        double y = hesapla(x);
   }
    return;
    static double hesapla(int x)
    {
          return Math.Sin(x) / 100 / 3.1416;
    }
```

Optimized source code:

```
for (int x = 1; x < 10000000; ++x)
{
        double y = Math.Sin(x) / 100 / 3.1416;
 }
    return true;
```

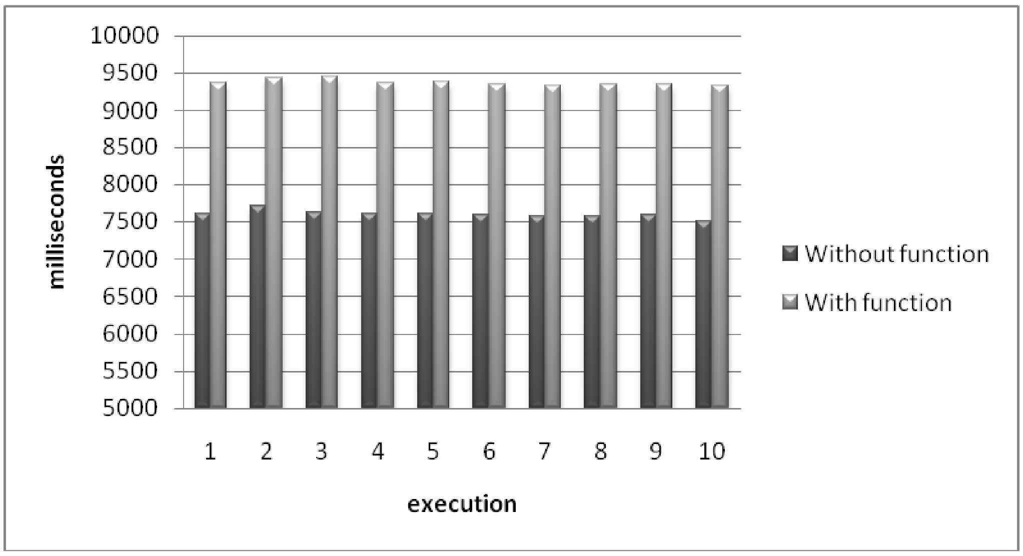


Figure 4.4 Function usage

When the loop counter is big enough, the energy counterpart of this style can be seen. In an example code, the total execution time and battery status change while using this style is as follows:

Using function: 54 minutes 7 seconds (%92 - %30)

Not using function: 46 minutes 42 seconds (%92 - %41)

4.5 Parameter Order

Parameter order in method calls in C# influences the speed. In a method, if some parameters are used more than others or in a tight loop, they should be put firstly. Because when you compile a method in C# language, the parameters are pushed into the stack and then that method uses the parameters from that stack. However, Microsoft compilers have an advanced optimization called 'fastcall', where the first two parameters in x86 are passed as registers (Allen, 2010). The speed of the code with the order of parameters changes as shown in Figure 4.5.

Original source code :

```
public int Method(int a,int b,int c,int d) {
        for (i = 1; i < 1000; i++) {
             d++; }
        return a+d;
}
```

Optimized code :

```
public int Method(int d,int b,int c,int a){
        for (i = 1; i < 1000; i++) {
             d++;       }
        return a+d;
}
```

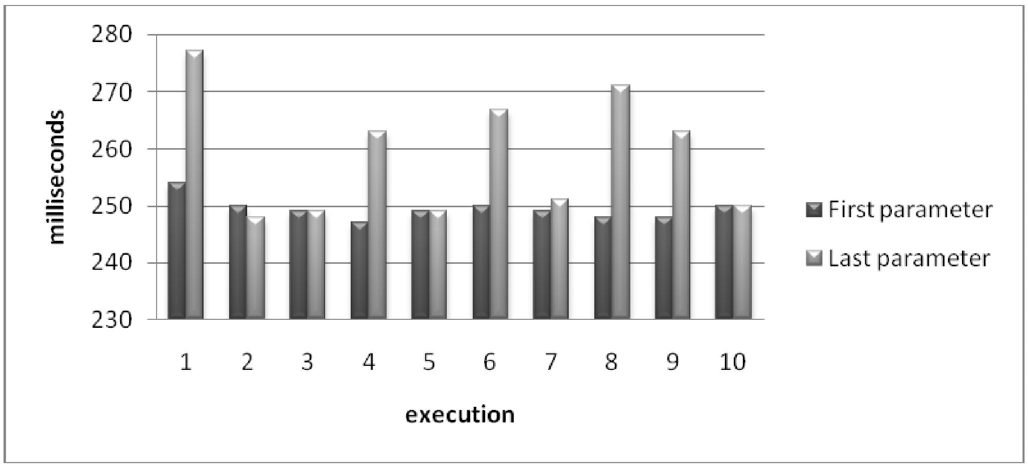


Figure 4.5 Parameter order

In fact this style's effect on the energy consumption cannot be seen clearly. Although using this style in a tight loop, the execution time and energy consumption of the source code do not change too much. As an example, in a tight loop the effect of this style is as follows:

Putting the mostly used parameter in the last order: 12 minutes 16 seconds (%99 –%84)

Putting the mostly used parameter in the first order: 11 minutes 21 seconds (%99 –%84)

4.6 ArrayList vs. Array

Depending on the workload and the usage in the application a wrong choice for the type could cost till 1000 times more energy.

Arrays are data structures to hold collections whose boundaries are static in which unused array elements cause unnecessary memory usage. ArrayLists can be defined as arrays whose size grow and shrink dynamically. Besides unnecessarily memory usage, it is inefficient in terms of time. Using arraylist in a tight loop instead of using array causes the code to execute slower as shown in Figure 4.6.

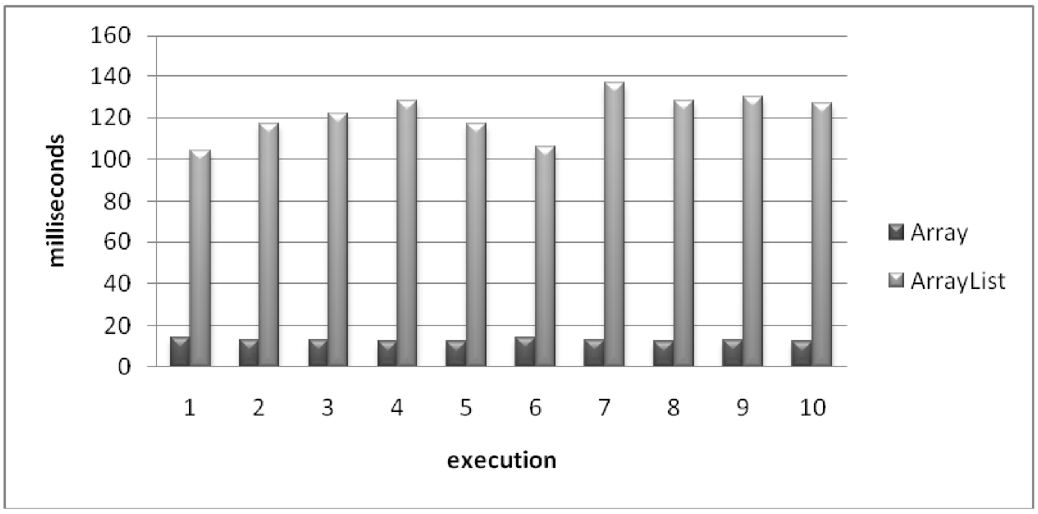


Figure 4.6 Array vs. Arraylist

Using arraylist instead of array also causes battery to decrease faster. The results of the test done to see this effect can be seen below:

Using arraylist: 6 minutes 23 seconds (%99 - % 91)

Using array: 2 minutes 58 seconds (%99 - %96)

4.7 Foreach vs. For

‘Foreach’ is used in C# instead of a for loop to simplify the code, but it is slower than a loop written using ‘For’. In fact foreach involves no performance penalty when used against arrays. However, when used against lists it involves the same overhead because in the background an enumerator is created and the loop is controlled within a try-catch block. Its effect can be seen in Figure 4.7.

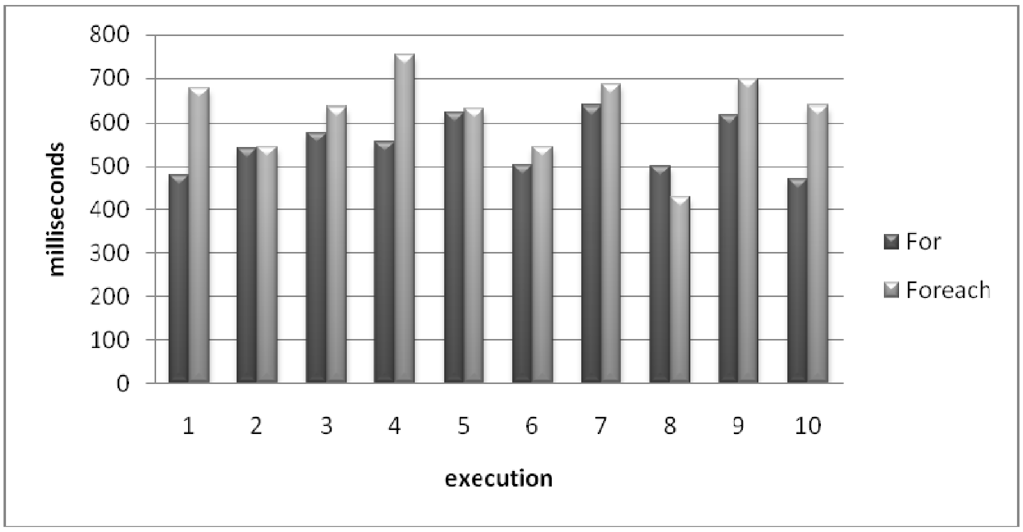


Figure 4.7 Foreach vs. For

Energy consumption counterpart of this style when using in a tight loop as an example are as follows:

Using “foreach”: 6 minutes 30 seconds (%85 - %76)

Using “for”: 6 minutes 1 seconds (%85 - %78)

4.8 String.Format vs. String Builder vs. Concatenation

Concatenating large strings in a loop is a performance drain and the StringBuilder's Append method is much more efficient. But the StringBuilder object requires a lot more memory than a String and it is not efficient for concatenating a small number of times. So it must be used if more than four concatenations are required.

Many .NET developers use the StringBuilder class whenever possible. However, it's not the fastest approach for concatenating small numbers of strings. Actually, any number can be combined in a single statement, although the performance benefit decreases above five or six substrings. This is due to instantiation and destruction overhead for the StringBuilder instance, as well as method-call overhead involved in calling Append() once for every added substring and ToString() once the string is built. The difference in terms of speed of the code can be seen in Figure 4.8. And battery usage test results are shown below.

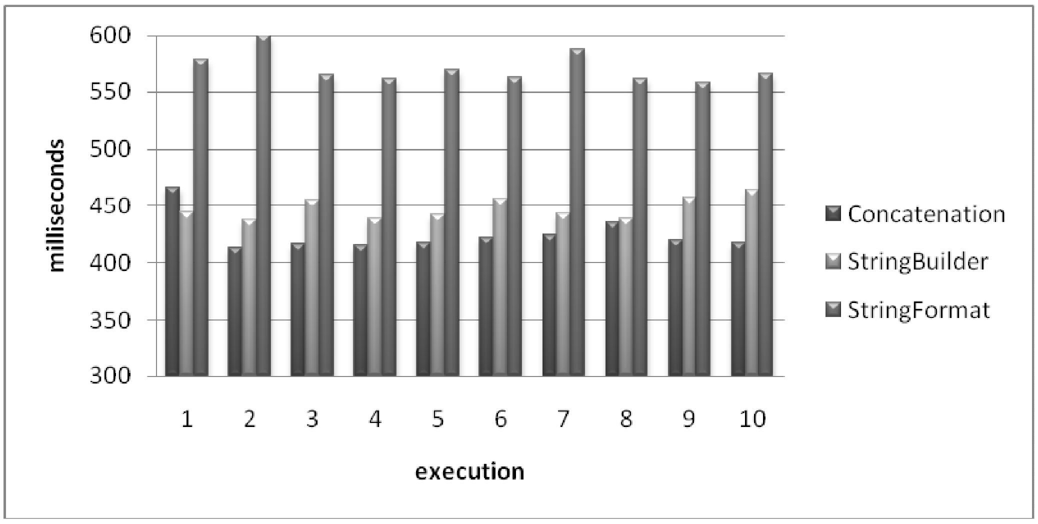


Figure 4.8 Concatenation vs. StringBuilder vs. StringFormat

Using StringFormat: 15 minutes 33 seconds (%99 - %82)

Using StringBuilder: 13 minutes 34 seconds (%99 - %84)

Using Concatenation: 13 minutes 2 seconds (%99 - %84)

4.9 Boxing-Unboxing

While working with object types boxing and unboxing are used. Boxing is the creation of a reference wrapper for a value type and unboxing is the extraction of the value type from the reference type. Boxing/unboxing enables value types to be treated as objects which are stored on the garbage collected heap. Whenever boxing is used, a new object is created on the managed heap and the value is copied in it. If it is done frequently, then lots of objects will be created and also the extra code will be executed for boxing and unboxing. Where possible this should be avoided as it is a major drain on performance especially, the overhead of both is most heavily felt in collection classes. The difference can be seen in Figure 4.9.

```
int i = 999;
object oObj = (object)i; // boxing
…
oObj= 999;
i = (int)oObj; // unboxing
```

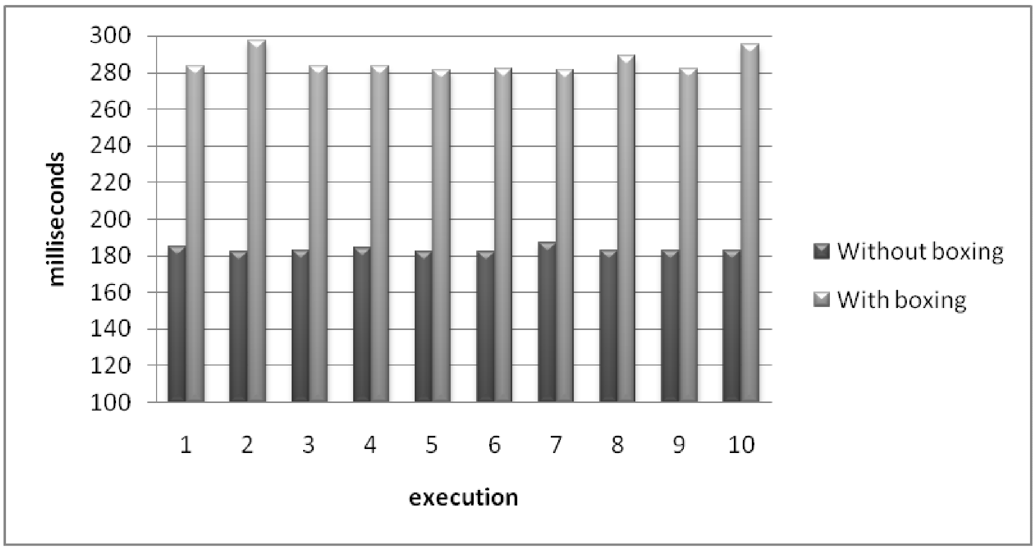


Figure 4.9 Boxing-unboxing

Near performance drain, using boxing and unboxing has an energy consumption penalty too. Its effect can be seen clearly from the result of the example execution of the test loop:

Using boxing/unboxing: 43 minutes 47 seconds (%99 - %44)

Using a specific type: 24 minutes 34 seconds (%99 - %68)

4.10 Reading Values of Objects Once

Reading values from objects is not as fast as assessing the value of a simple variable. So if a value of an object will be used multiple times especially in loops, its value must be read for once at the beginning and then that variable should be accessed when needed. Figure 4.10 shows the effect of this strategy.

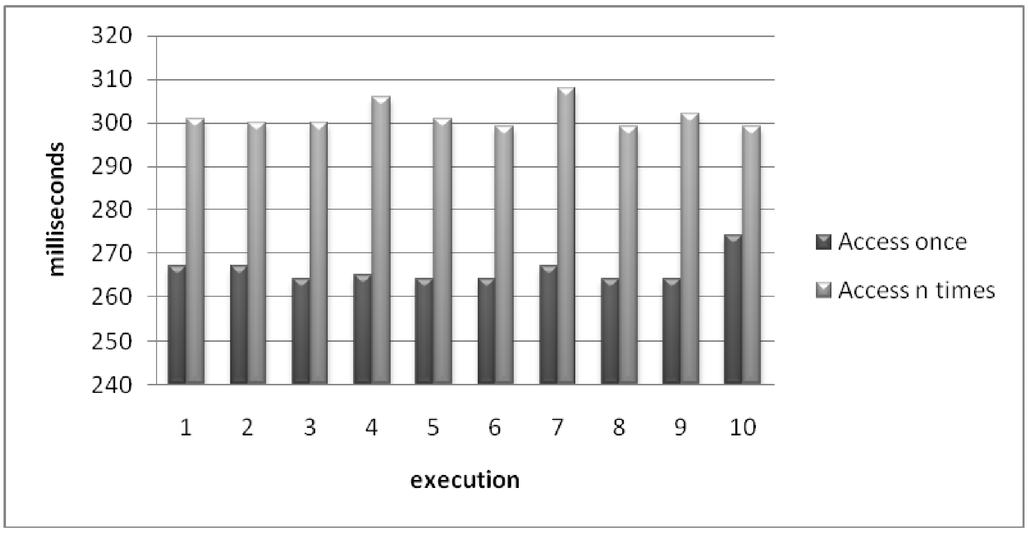


Figure 4.10 Reading values of objects once vs. n-times

Reading values of objects is expensive in terms of energy and battery usage as its effect can be seen in very big loops. One of the results is as follows:

Reading value of an object for n-times: 48 minutes 16 seconds (%92 - %30)

Reading value of an object for once: 46 minutes 2 seconds (%92 - %31)

4.11 Special Operators

There are special operators that enable to do math operations in a more compact way. Using these special operators efficiently may help compilers to produce code more efficient.

```
Original source code:  a = a + b;
                       b = b + 1;
Optimized code :       a = a + b++;
```

As you see, in the first way, b variable will be stored in register twice (one for addition and one for increment). But in the second way, it will be stored for once. This supplies smaller and faster program as shown in Figure 4.11.

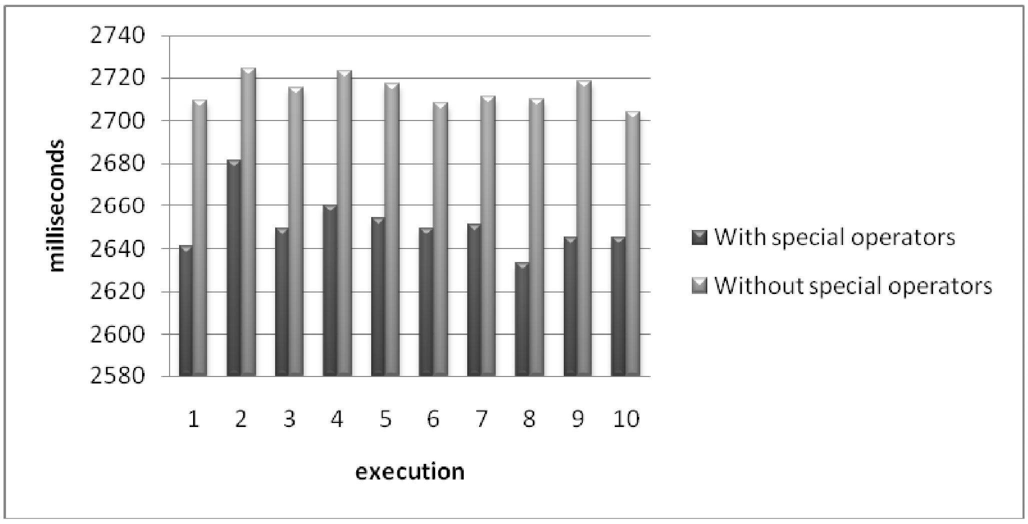


Figure 4.11 Special operators

If you increase the counter of the test loop, this style's effect on energy consumption appears. For example in a tight loop, execution time of the code and change amount in battery status becomes as follows:

Without special operators: 33 minutes 59 seconds (%94 - %50)

With special operators: 32 minutes 42 seconds (%94 - %52)

4.12 Parallel Programming

Multi-core machines are now becoming standard with the need of programs which run faster and consume less energy. The key to performance improvements is therefore to run a program on multiple processors in parallel. But it is still very hard to write algorithms that actually take advantage of those multiple processors. Despite running on a multi-core machine, most applications use a single core and see no speed improvement. So programs must be written in a new way named 'parallel programming'. Figure 4.12 shows the effect of this new way on the speed of our programs.

Original source code:

```
for (int i = 0; i < 100; i++) {
        a[i] = a[i]*a[i];
}
```

Optimized source code *(With parallel programming):*

```
Parallel.For(0, 100, delegate(int i) {
         a[i] = a[i]*a[i];
  });
```

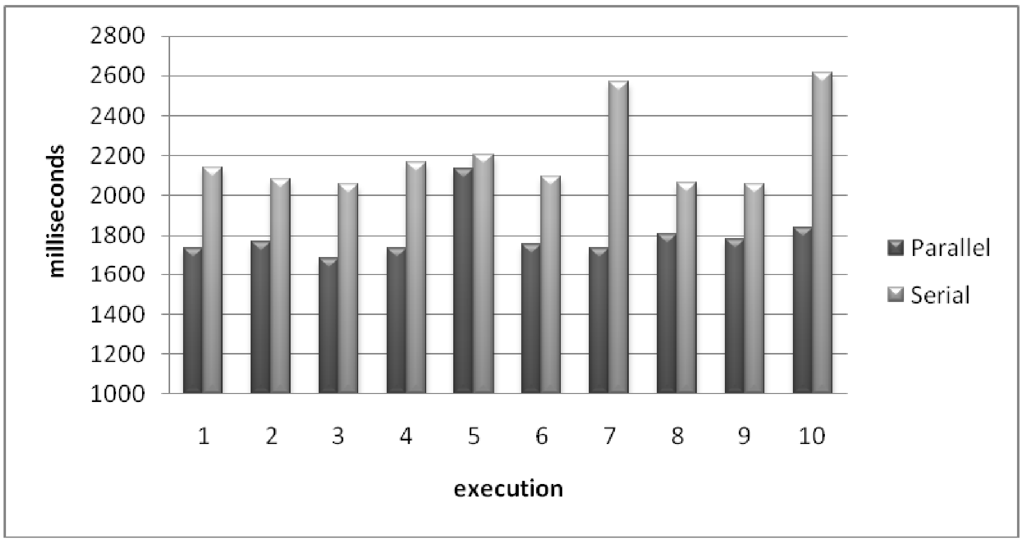


Figure 4.12 Parallel vs. Serial programming

Energy consumption counterpart of this style when using in a tight loop as an example are as follows:

With serial programming: 52 minutes 11 seconds (%99 - %32)

With parallel programming: 43 minutes 2 seconds (%99 - %44)

Here, it must be noted that using more thread increases cpu utilization for finishing the job faster but it does not cause more energy consumption as results show.

4.13 Smart Try-Catch - Minimize Exceptions

Catching and throwing exceptions is very expensive and should be avoided where possible. For example exception blocks should never be used to catch an error caused by attempting to access a null object, instead a statement should be used to test if the object is null before accessing it. Figure 4.13 shows the effect of this choice on the performance of our programs:

Original source code:

```
try {
  //perform operation
}
Catch {
  //catch error
}
```

Optimized code :

```
if (myObj != null){
  //perform operation
}
else {
  //catch error
}
```

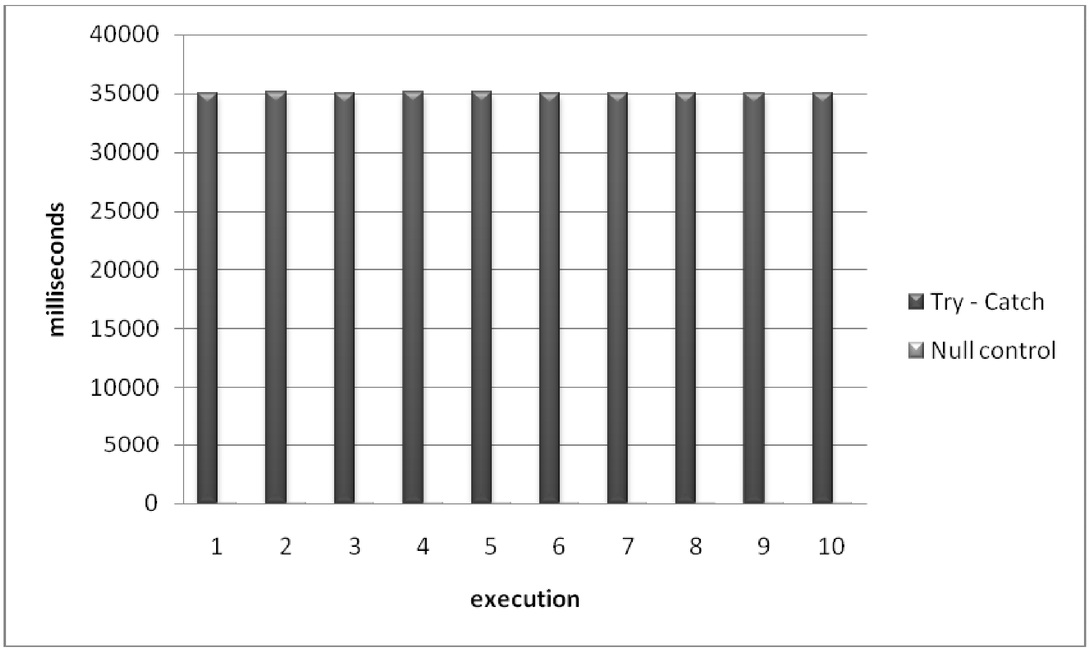


Figure 4.13 Using Try-Catch vs. Control

Using try-catch blocks instead of using statements to prevent an error has a very important effect in terms of energy near performance. This effect can be seen from the example test results below (the difference gets bigger as exception cases increase):

Using try-catch: 54 minutes 3 seconds (%99 - %39)
Using control statements: 7 minutes 21 seconds (%99 - %89)

The techniques analyzed in this section can be summarized as shown in Table 4.1 and Table 4.2.

Table 4.1 Summary of optimization techniques

Strategy 1	**Strategy 2**	**Strategy 3**	**Recommendation**	**Environment**
Use Class	Use Struct	*	Use Struct	OOP (C#)
Use Static Variable	Use Dynamic Variable	*	Use Static Variable	OOP (C#)
Use Recursion	Use Iteration	*	Use Iteration	OOP (C#)
Use function	Not use function	*	Not use function for sometimes	OOP (C#)
Use mostly used parameters in the first order	Use mostly used parameters in the last order	*	Use mostly used parameters in the first order	OOP (C#)
Use Arraylist	Use Array	*	Use Array	OOP (C#)
Use Foreach	Use For	*	Use For	OOP (C#)
Use StringFormat	Use StringBuilder	Use Concatenation	Use StringFormat	OOP (C#)
Use boxing	Not use boxing	*	Not use boxing	OOP (C#)

Strategy 1	Strategy 2	Strategy 3	Recommendation	Environment
Read values of objects once	Read values of objects more	*	Read values of objects once	OOP (C#)
Use special operators	Use basic operators	*	Use special operators efficiently	OOP (C#)
Parallel programming	Serial programming	*	Parallel programming	OOP (C#)
Use try-catch	Not use try-catch	*	Not use try-catch	OOP (C#)
Use events/notification	Use polling	*	Use events/notification	General
Use Balanced multithreading	Use Unbalanced multithreading	Use Single threading	Use Balanced multithreading	General
Use shorter timer intervals	Use longer timer intervals	*	Use shorter timer intervals	General
Use loops	Use loop unrolling if possible	*	Use loop unrolling if possible	General
Use Big functions	Use Short functions	*	Use Short functions	General
Use Complex algorithms	Use Simple algorithms	*	Use Simple algorithms	General
Use animations	Not use animations	*	Not use animations	General

* : Does not have third strategy in the same concept

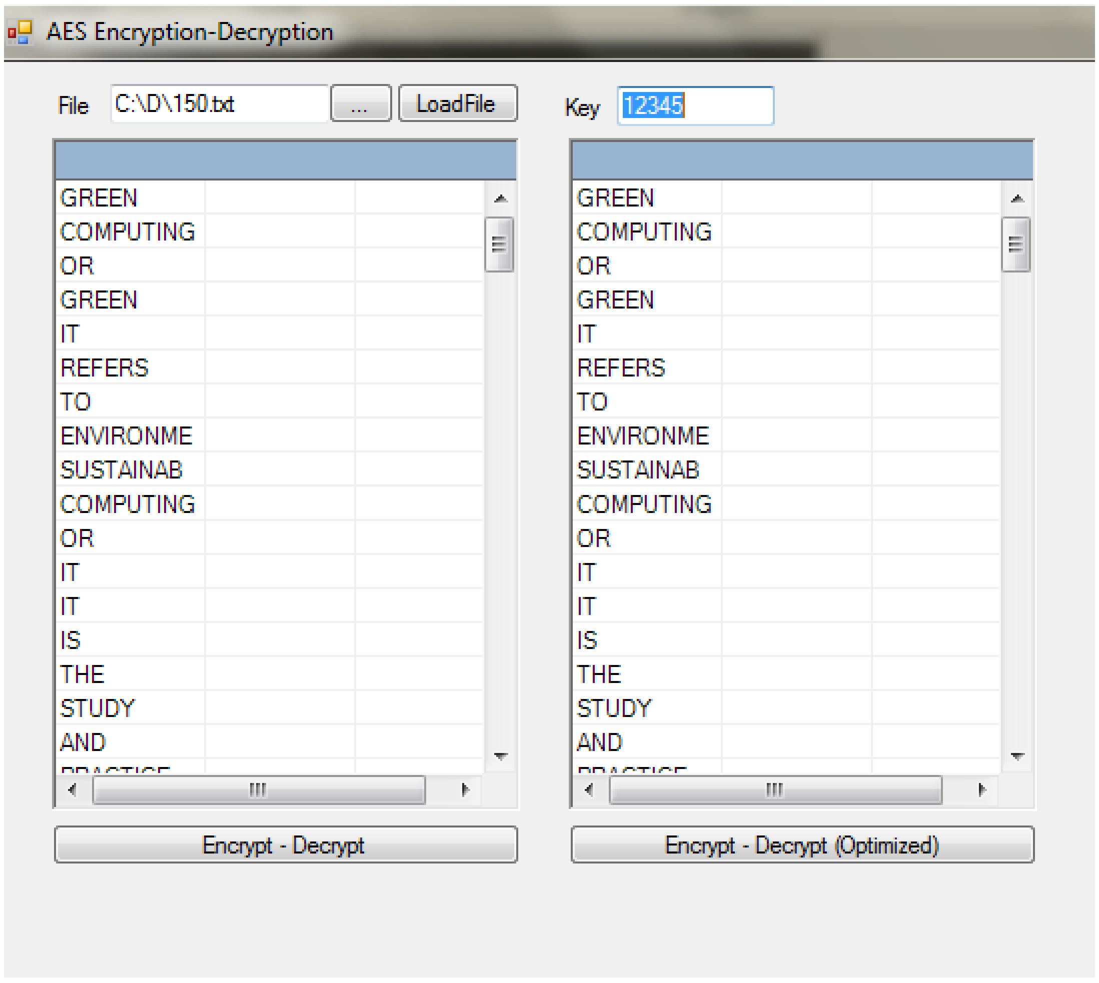
Table 4.2 Test results of optimization techniques

Strategy 1	**Battery status change**	**Execution time**	**Strategy 2**	**Battery status change**	**Execution time**
Using Class	%98-%82	792 sec.	Using Struct	%98-%85	634 sec.
Using Dynamic variable	%93-%64	1348 sec.	Using Static variable	%93-%67	1260 sec.
Using Recursion	%99-%94	302 sec.	Using Iteration	%99-%97	154 sec.
Using function	%92-%30	3247 sec.	Not using function	%92-%41	2802 sec.
Parameter in last order	%99-%84	736 sec.	Parameter in first order	%99-%84	681 sec.
Using Arraylist	%99-%91	383 sec.	Using Array	%99-%96	178 sec.
Using Foreach	%85-%76	390 sec.	Using For	%85-%78	361 sec.
Using StringFormat	%99-%82	933 sec.	Using Concatenation	%99-%84	782 sec.
Using boxing/unboxi ng	%99-%44	2627 sec.	Using specific type	%99-%68	1474 sec.
Reading value of an object for n-times	%92-%30	2896 sec.	Reading value of an object for once	%92-%31	2762 sec.
Using regular operators	%94-%50	2039 sec.	Using special operators	%94-%52	1962 sec.
Using Serial programming	%99-%32	3131 sec.	Using Parallel programming	%99-%44	2582 sec.
Using try-catch	%99-%39	3243 sec.	Using control statements	%99-%89	441 sec.

CHAPTER FIVE

DEVELOPMENT & TEST RESULTS

As an example an application has been developed to see the effect of the strategies above. In this application there is a form in which a file containing the list of words can be chosen and there are two different buttons to start to encrypt and decrypt them in a loop. Figure 5.1 shows a screenshot of the form. First button triggers a class implementation which uses the worst ways and types versus the second one uses the best choices for source code optimization.


AES Encryption-Decryption

File C:\D\150.txt ... LoadFile

Key 12345

GREEN		
COMPUTING		
OR		
GREEN		
IT		
REFERS		
TO		
ENVIRONME		
SUSTAINAB		
COMPUTING		
OR		
IT		
IT		
IS		
THE		
STUDY		
AND		

GREEN		
COMPUTING		
OR		
GREEN		
IT		
REFERS		
TO		
ENVIRONME		
SUSTAINAB		
COMPUTING		
OR		
IT		
IT		
IS		
THE		
STUDY		
AND		

Encrypt - Decrypt

Encrypt - Decrypt (Optimized)

Figure 5.1 Choosing data to work on.

The differences can be summarized as:

- A type formed for holding the words and their encrypted and decrypted states. In the first class these object types are hold in an object list and in the second one they are hold in a list which will not require any boxing-unboxing operation.

Original :

```
private List<object> _listData;
```

Optimized:

```
private List<InputData> _listDataOptimized;
```

- Word count is needed in different steps of the program. In the first class, this value is calculated by the length property of the word-list collection and in the second one the collection’s length property is read into a variable and that variable is used where needed.

Original:

```
if (counter == (_listData.Count % 2 == 0 ?
_listData.Count / 2 : (_listData.Count - 1) / 2)) { … }
```

Optimized:

```
_wordCount = _listData.Count;
if (counter == (_wordCount % 2 == 0 ? _wordCount / 2 :
(_wordCount - 1) / 2))  {
   …
   }
```

- In the optimized one, the words are encrypted and decrypted in parallel while the first one does the same operations in serial.

Original:

```
   foreach (object dataObj in _listData)  {
        …
encryptedStr           =           EncryptStr(data.Ad,
key.ToString(), 0);
}
```

Optimized:

```
  Parallel.ForEach<InputData>(_listData,       s       =>
EncryptParallel(s, ref counter, ref tempToplam, key));
```

- In the first one, encryption and decryption methods are recursive where it is optimized by using iteration in the second one.

Original:

```
  private  static  string  EncryptStr(string  str,  string
key, int counter) {
  if (counter < 20) {
     counter = counter + 1;
  str = _aes.Encrypt(EncryptStr(str, key, counter), key,
"", "MD5", 3, "16CHARSLONG12345", 128);
    }
    return str;
   }
```

Optimized:

```
  private  static  string  EncryptStr(int  counter,  string
str, string key)  {
  for (int i = 0; i < 20; i++) {
  str   =   _aes.Encrypt(str,   key,   "",   "MD5",   3,
"16CHARSLONG12345", 128);
  }
  return str;
  }
```

- In the original one for mathematical operations normal operators are used, but in the optimized one special operators are used in an efficient way.

Original:

```
    tempToplam = tempToplam + counter;
    counter = counter + 1;
```

Optimized:

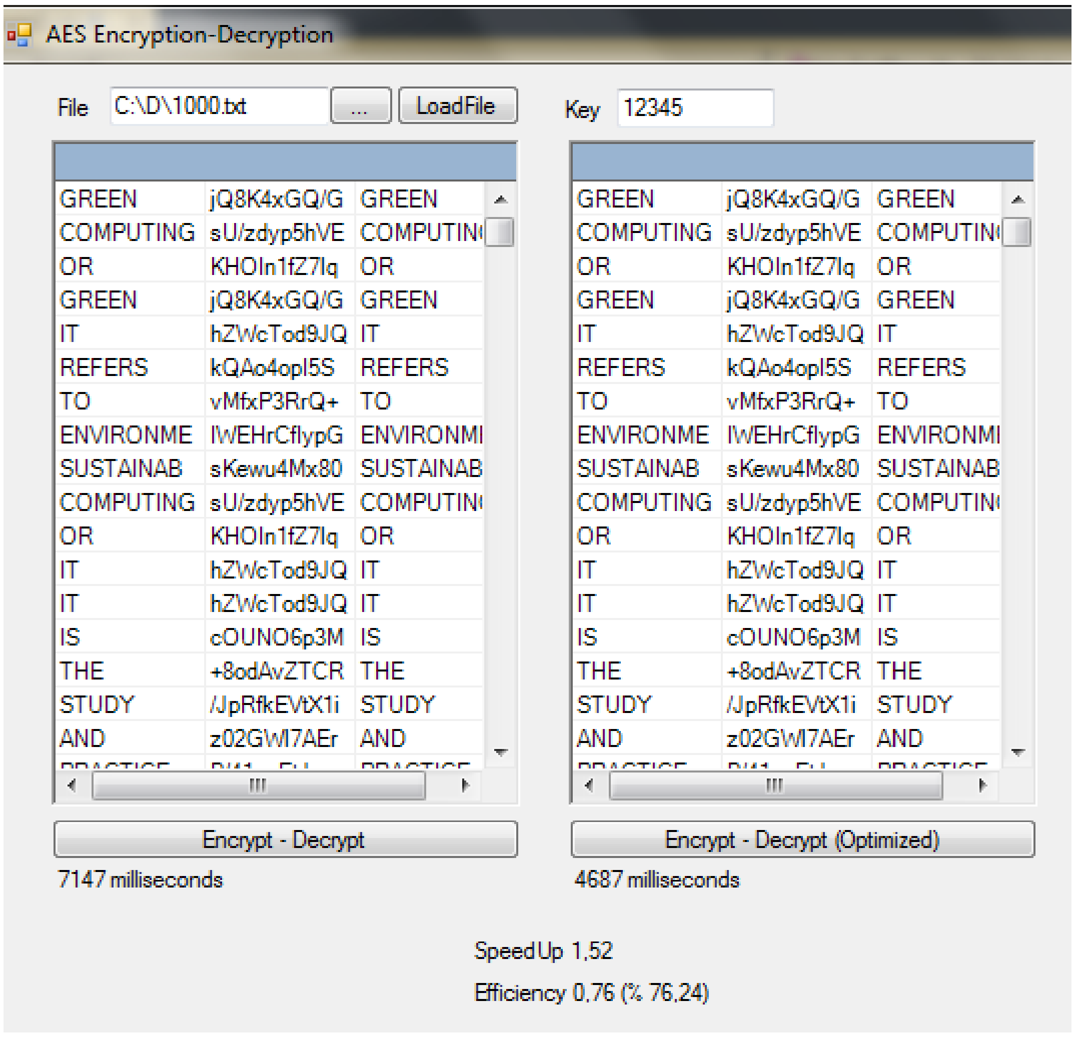
```
  tempToplam = tempToplam + counter++;
```

The results that can be seen in Figure 5.2 show that, they are giving the same outputs which mean they do the same job but their execution times are very different as seen in the figure below so as the energy they consume. After finding total time results, the efficiency and speed up values are calculated by using Equation 2 and Equation 3 (Şenyurt, 2010). Ts is the time taken to run the code serial and Tp is the time taken to run parallel algorithm on N processors.

SpeedUp = SN = Ts / Tp (2)

Efficiency = EN = SN / N (3)

This shows us that, by choosing convenient ways, appropriate strategies and using true types we can write faster and more efficient programs without doing any hardware changes.


AES Encryption-Decryption

File C:\D\1000.txt ... LoadFile

Key 12345

GREEN	jQ8K4xGQ/G	GREEN
COMPUTING	sU/zdyp5hVE	COMPUTIN
OR	KHOIn1fZ7lq	OR
GREEN	jQ8K4xGQ/G	GREEN
IT	hZWcTod9JQ	IT
REFERS	kQAo4opl5S	REFERS
TO	vMfxP3RrQ+	TO
ENVIRONME	lWEHrCflypG	ENVIRONMI
SUSTAINAB	sKewu4Mx80	SUSTAINAB
COMPUTING	sU/zdyp5hVE	COMPUTIN
OR	KHOIn1fZ7lq	OR
IT	hZWcTod9JQ	IT
IT	hZWcTod9JQ	IT
IS	cOUNO6p3M	IS
THE	+8odAvZTCR	THE
STUDY	/JpRfkEVtX1i	STUDY
AND	z02GWI7AEr	AND

GREEN	jQ8K4xGQ/G	GREEN
COMPUTING	sU/zdyp5hVE	COMPUTIN
OR	KHOIn1fZ7lq	OR
GREEN	jQ8K4xGQ/G	GREEN
IT	hZWcTod9JQ	IT
REFERS	kQAo4opl5S	REFERS
TO	vMfxP3RrQ+	TO
ENVIRONME	lWEHrCflypG	ENVIRONMI
SUSTAINAB	sKewu4Mx80	SUSTAINAB
COMPUTING	sU/zdyp5hVE	COMPUTIN
OR	KHOIn1fZ7lq	OR
IT	hZWcTod9JQ	IT
IT	hZWcTod9JQ	IT
IS	cOUNO6p3M	IS
THE	+8odAvZTCR	THE
STUDY	/JpRfkEVtX1i	STUDY
AND	z02GWI7AEr	AND

Encrypt - Decrypt

7147 milliseconds

Encrypt - Decrypt (Optimized)

4687 milliseconds

SpeedUp 1,52

Efficiency 0,76 (% 76,24)

Figure 5.2 One of the results of the test

Figure 5.3 shows the results of ten consecutive executions of the program and the difference between the original and optimized code.

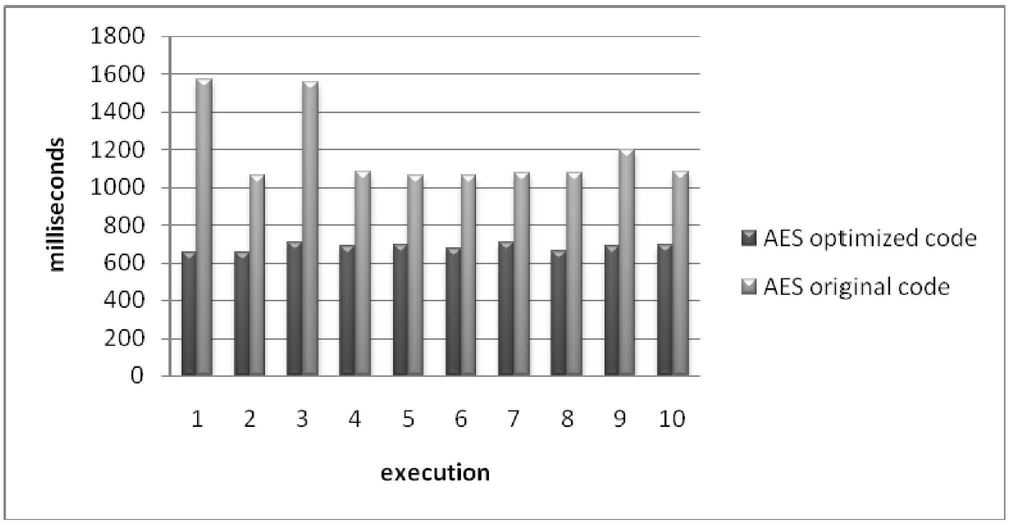


Figure 5.3 Results of comparing the original and optimized code

The input data makes this difference bigger as its size becomes larger. Besides the changes on input size, changes in hardware design effect the speed and CPU usage of code too. These effects have been observed in different machines and with inputs with different sizes, and the results can be seen in the following figures tested on different machines.

5.1 Dual Core Machine – 2 threads

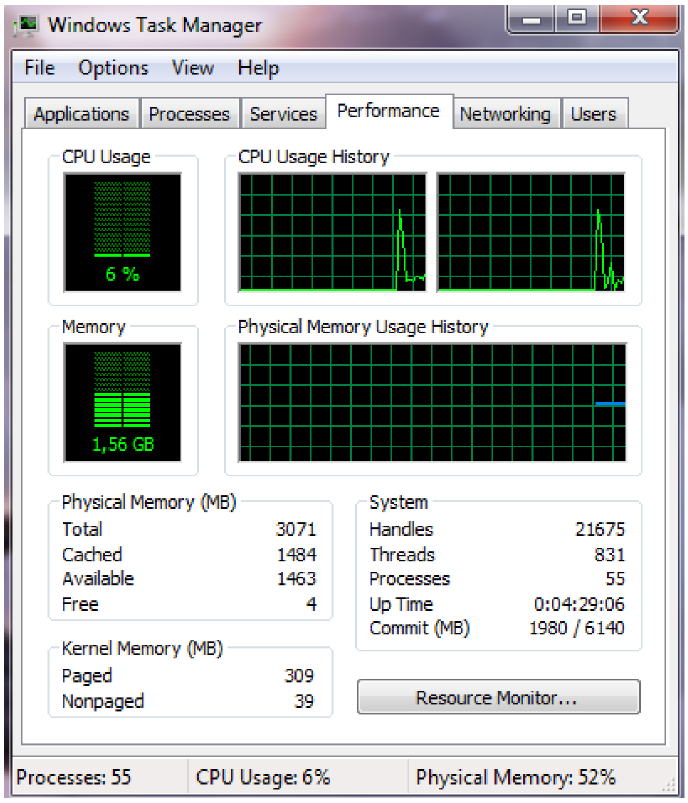
This hardware design can be summarized as:

"Processor : Intel Core 2 Duo CPU – T6600 2.20 GHz"

"Memory: 3 GB RAM"

"System type: 32 bit Operating System"

The snapshot of the processors can be seen in Figure 5.4.


Windows Task Manager

File Options View Help

Applications | Processes | Services | Performance | Networking | Users

CPU Usage: 6 %

Memory: 1,56 GB

Physical Memory (MB)

Total	3071
Cached	1484
Available	1463
Free	4

Kernel Memory (MB)

Paged	309
Nonpaged	39

System

Handles	21675
Threads	831
Processes	55
Up Time	0:04:29:06
Commit (MB)	1980 / 6140

Processes: 55 CPU Usage: 6% Physical Memory: 52%

Figure 5.4 Snapshot of processors in dual core machine

On this hardware design, the effect of the input size on the execution time and CPU usage of the code can be seen in figures below.

5.1.1 50 words

When the input file contains 50 words, the original code runs for about 20 seconds (0-20) with 60 percent of the CPU and the optimized code runs for about 13 seconds (20-33) with about 100 percent of the CPU as can be seen in Figure 5.5.

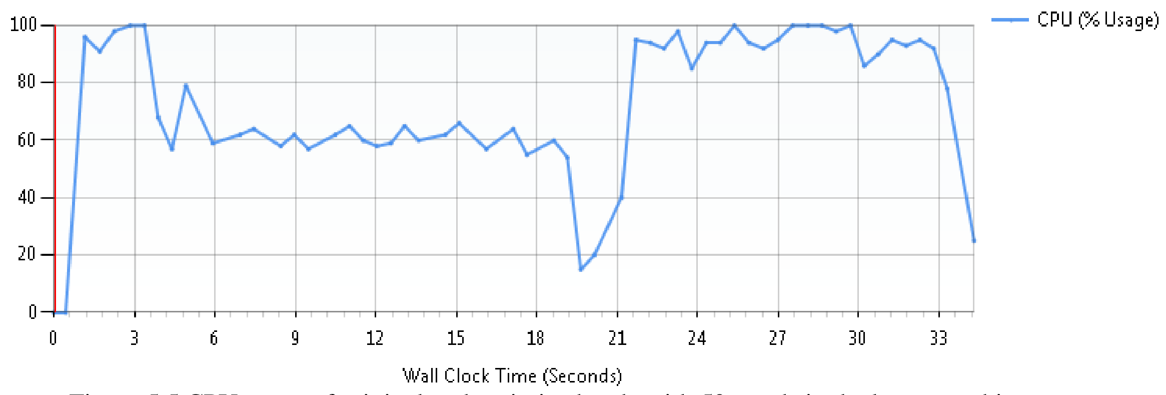


Figure 5.5 CPU usage of original and optimized code with 50 words in dual core machine

5.1.2 100 words

When the input file contains 100 words, the original code runs for about 40 seconds (0-40) with 60 percent of the CPU and the optimized code runs for about 23 seconds (40-63) with about 100 percent of the CPU as shown in Figure 5.6.

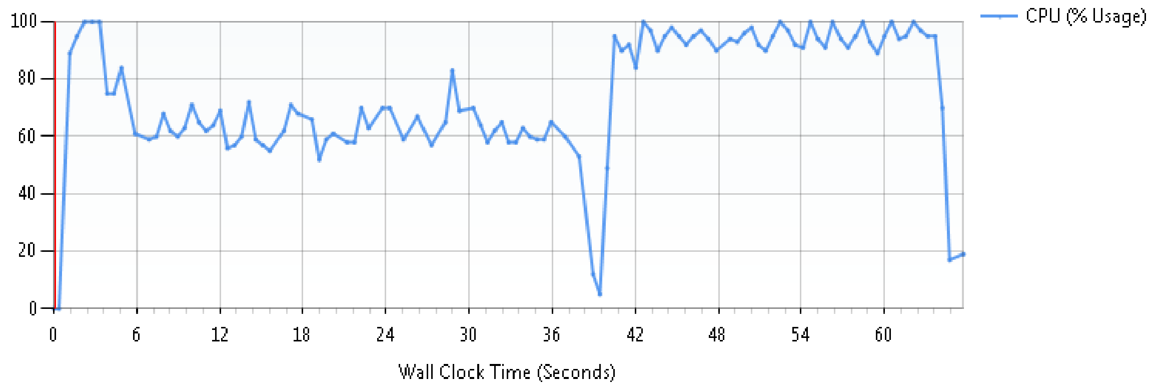


Figure 5.6 CPU usage of original and optimized code with 100 words in dual core machine

5.1.3 150 words

When working with 150 words, the original code runs for about 54 seconds (0-54) with 60 percent of the CPU and the optimized code runs for about 38 seconds (54-92) with about 100 percent of the CPU as shown in Figure 5.7.

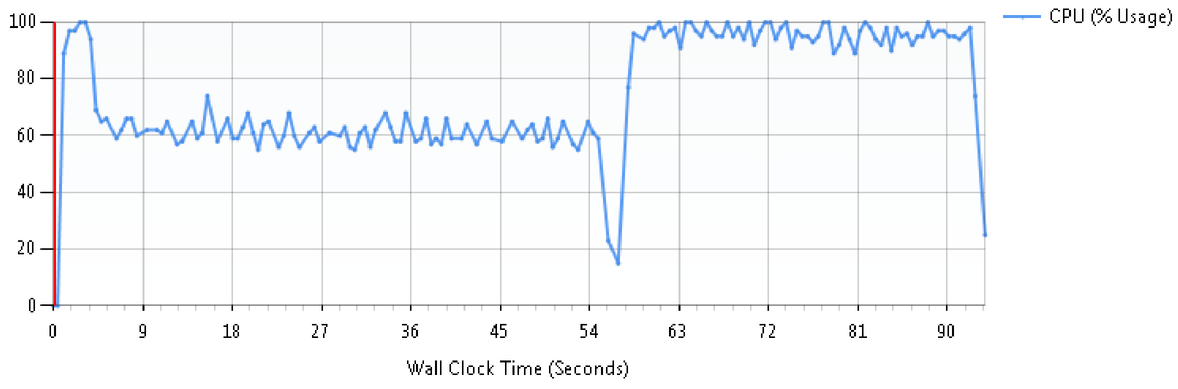


Figure 5.7 CPU usage of original and optimized code with 150 words in dual core machine

5.1.4 300 words

When working with 300 words, the original code runs for about 112 seconds (0-112) with 60 percent of the CPU and the optimized code runs for about 70 seconds (112-182) with about 100 percent of the CPU as shown in Figure 5.8.

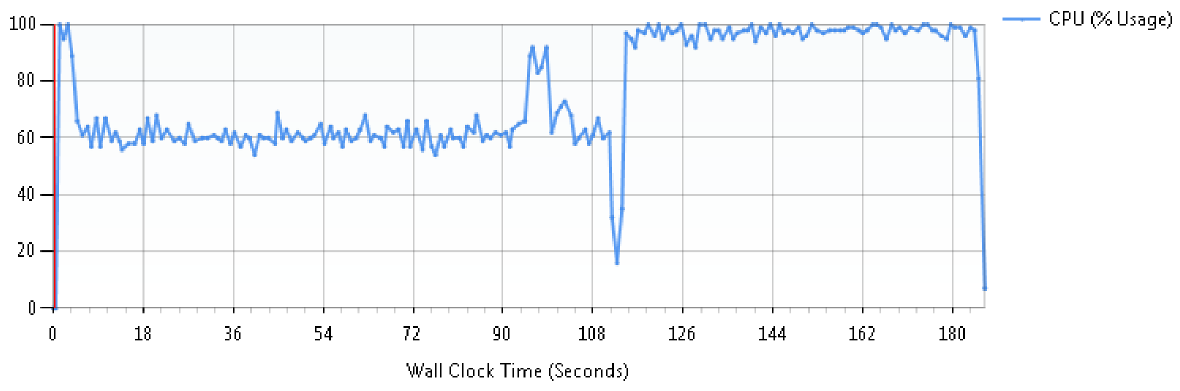


Figure 5.8 CPU usage of original and optimized code with 300 words in dual core machine

5.1.5 600 words

When 600 words are used during the program, the original code runs for about 220 seconds (0-220) with 60 percent of the CPU and the optimized code runs for about 140 seconds (220-360) with about 100 percent of the CPU as shown in Figure 5.9.

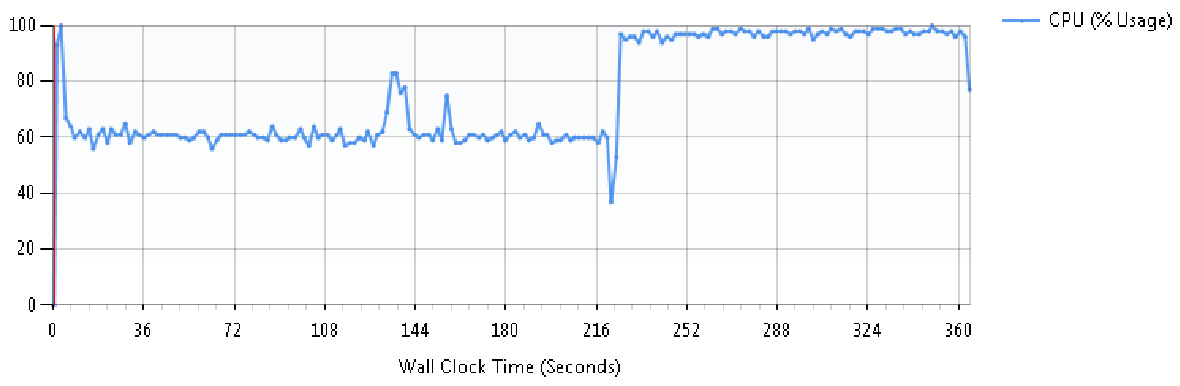


Figure 5.9 CPU usage of original and optimized code with 600 words in dual core machine

5.1.6 1000 words

When the input size is increased to 1000 words, the original code runs for about 356 seconds (0-356) with 60 percent of the CPU and the optimized code runs for about 224 seconds (356-580) with about 100 percent of the CPU. These results can be seen in Figure 5.10.

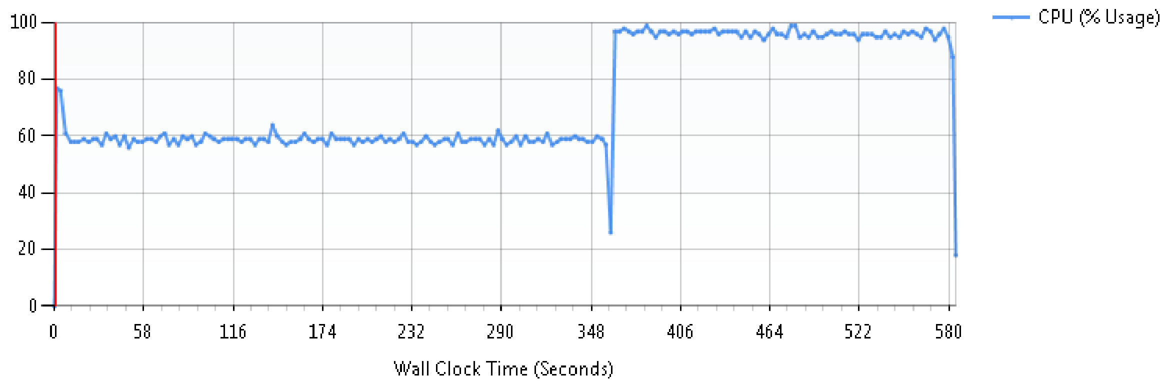


Figure 5.10 CPU usage of original and optimized code with 1000 words in dual core machine

5.1.7 5000 words

When the input size is increased to 5000 words, the original code runs for about 1770 seconds (0-1770) with 60 percent of the CPU and the optimized code runs for about 1180 seconds (1770-2950) with about 100 percent of the CPU as shown in Figure 5.11.

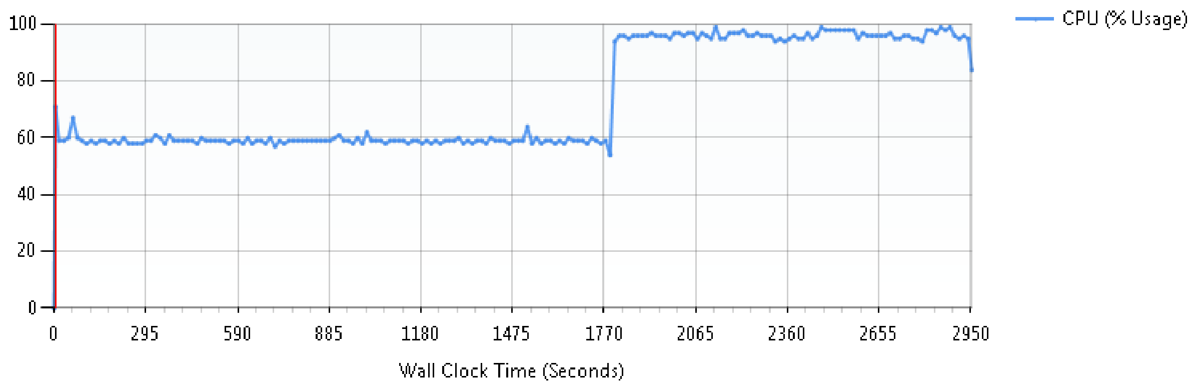


Figure 5.11 CPU usage of original and optimized code with 5000 words in dual core machine

5.1.8 10000 words

Lastly when the input size is increased to 10000 words, the original code runs for about 3670 seconds (0-3670) with 60 percent of the CPU and the optimized code runs for about 2450 seconds (3670-6120) with about 100 percent of the CPU as can be seen in Figure 5.12.

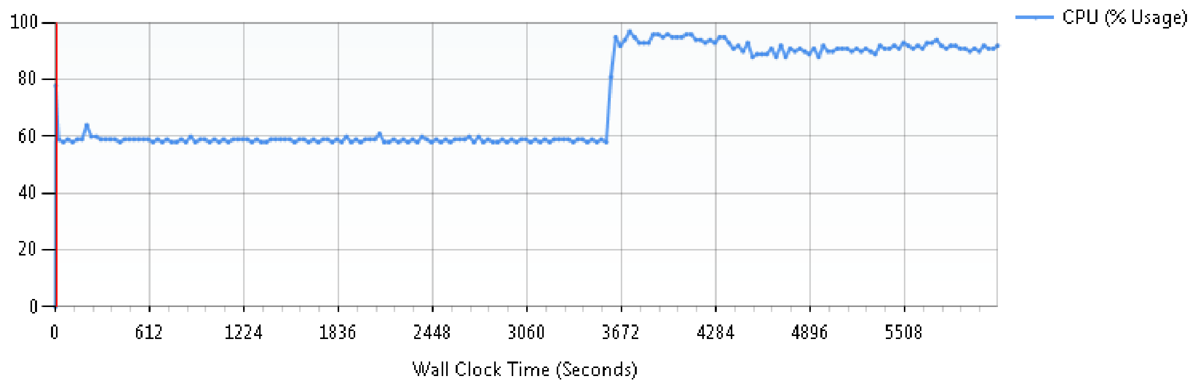


Figure 5.12 CPU usage of original and optimized code with 10000 words in dual core machine

5.2 Two Quad Core Machine

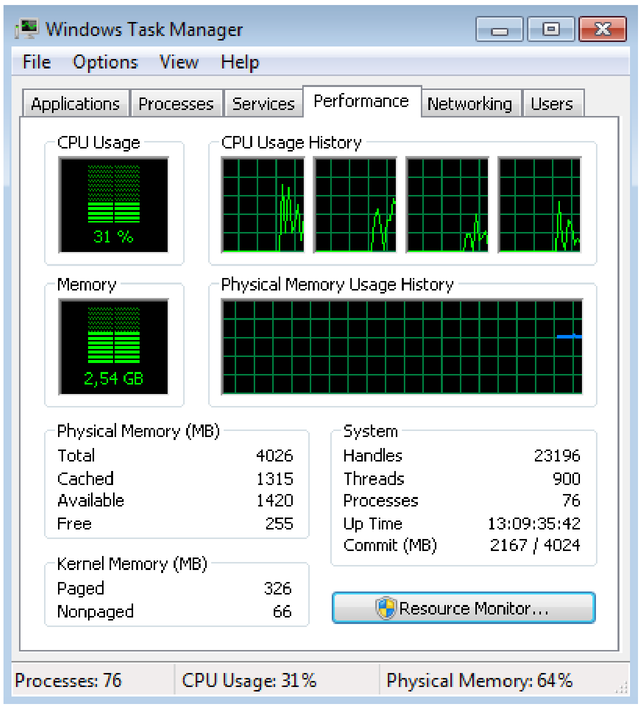
This hardware design can be summarized as:

"Processor : Intel Core 2 Quad CPU – Q6600 2.40 GHz"

"Memory: 4 GB RAM"

"System type: 64 bit Operating System"

The design of the processors can be seen in Figure 5.13.



Physical Memory (MB)
Total 4026
Cached 1315
Available 1420
Free 255

Kernel Memory (MB)
Paged 326
Nonpaged 66

System
Handles 23196
Threads 900
Processes 76
Up Time 13:09:35:42
Commit (MB) 2167 / 4024

Processes: 76 CPU Usage: 31% Physical Memory: 64%

Figure 5.13 Snapshot of processors in 2 quad core machine

The effect of the input size on CPU usage with this design can be seen in the following figures:

5.2.1 50 words

When working with 50 words, the original code runs for about 16 seconds (0-16) with 30 percent of the CPU and the optimized code runs for about 6 seconds (18-24) with about 90 percent of the CPU as shown in Figure 5.14.

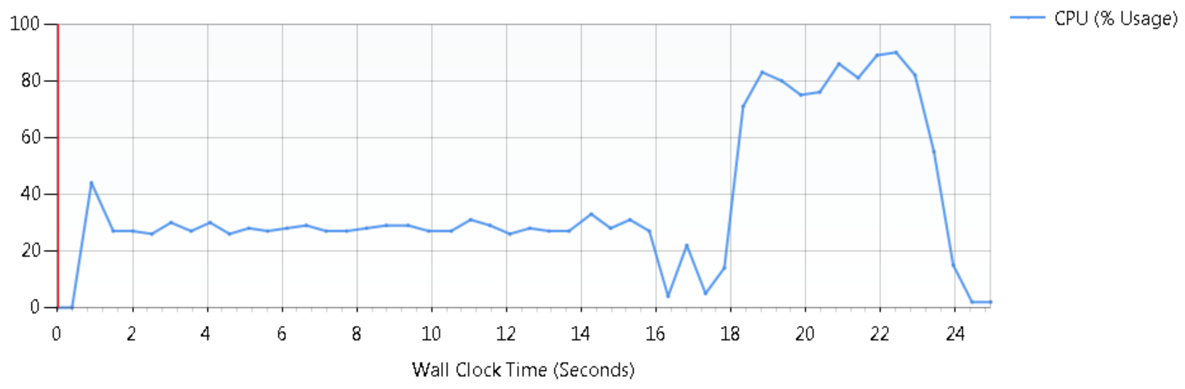


Figure 5.14 CPU usage of original and optimized code with 50 words in 2 quad core machine

5.2.2 100 words

When working with 100 words, the original code runs for about 32 seconds (0-32) with 30 percent of the CPU and the optimized code runs for about 12 seconds (32-44) with about 90 percent of the CPU as shown in Figure 5.15.

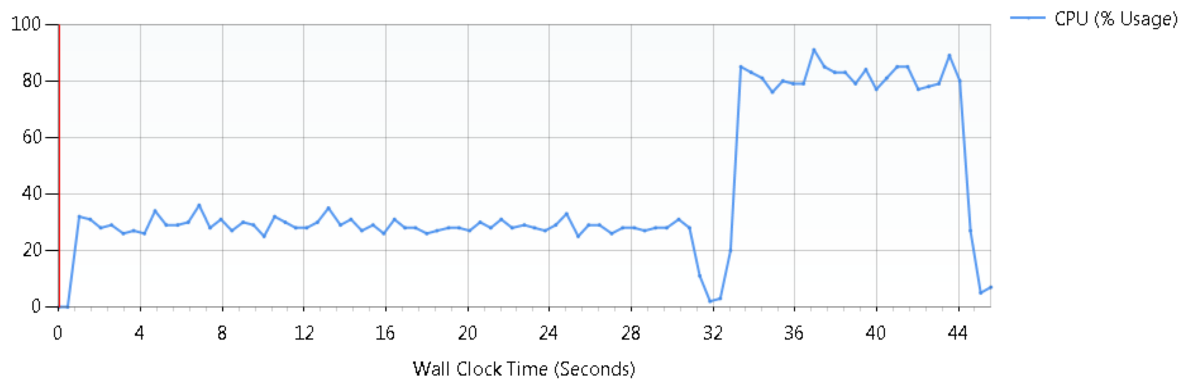


Figure 5.15 CPU usage of original and optimized code with 100 words in 2 quad core machine

5.2.3 150 words

When working with 150 words, the original code runs for about 48 seconds (0-48) with 30 percent of the CPU and the optimized code runs for about 18 seconds (48-66) with about 90 percent of the CPU as shown in Figure 5.16.

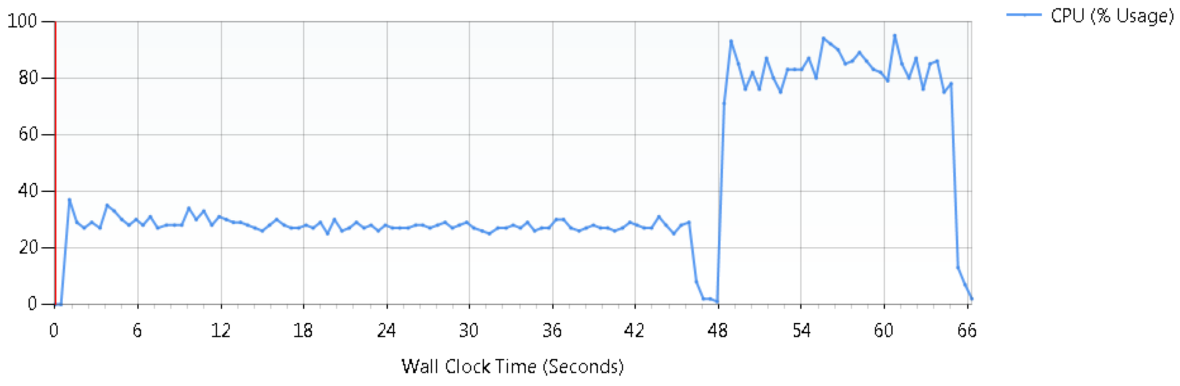


Figure 5.16 CPU usage of original and optimized code with 150 words in 2 quad core machine

5.2.4 300 words

Figure 5.17 shows that when the input file contains 300 words, the original code runs for about 94 seconds (0-94) and the optimized code runs for about 36 seconds (94-130).

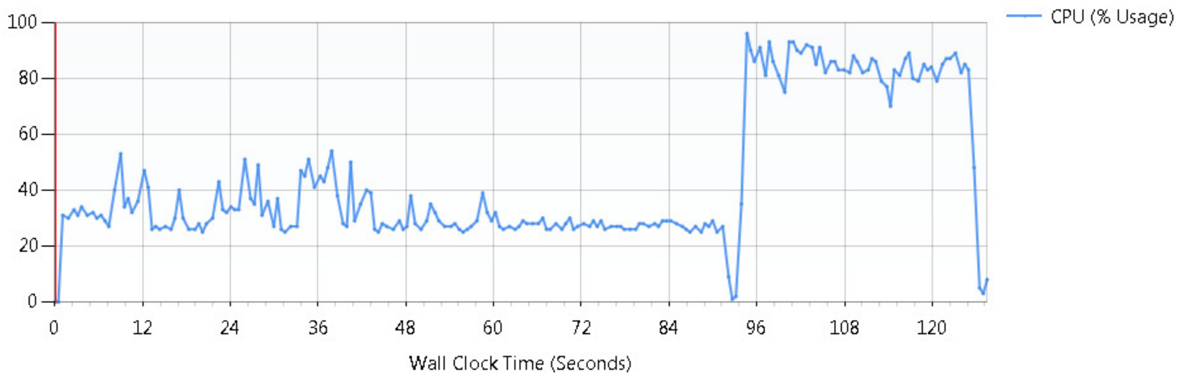


Figure 5.17 CPU usage of original and optimized code with 300 words in 2 quad core machine

5.2.5 600 words

As shown in Figure 5.18, when the input file contains 600 words, the original code runs for about 185 seconds (0-185) and the optimized code runs for about 65 seconds (185-250).

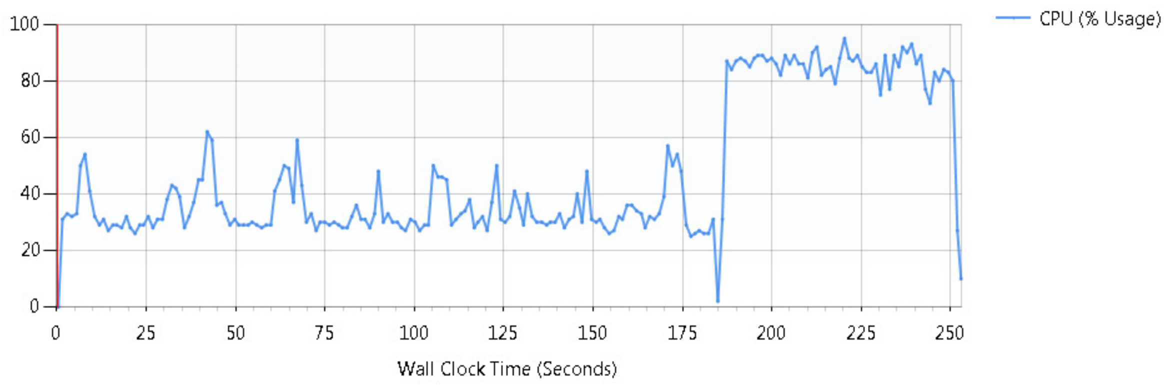


Figure 5.18 CPU usage of original and optimized code with 600 words in 2 quad core machine

5.2.6 1000 words

When the input file contains 1000 words, the original code runs for about 305 seconds (0-305) and the optimized code runs for about 100 seconds (305-405) like shown in Figure 5.19.

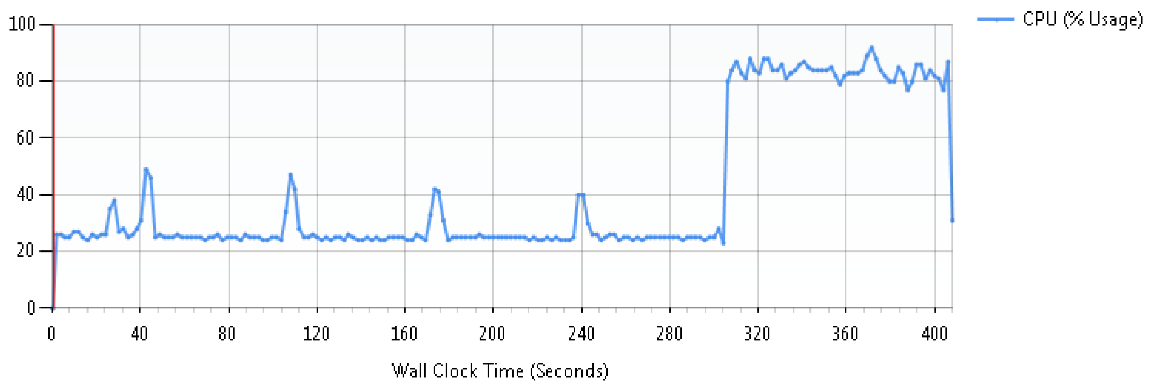


Figure 5.19 CPU usage of original and optimized code with 1000 words in 2 quad core machine

5.2.7 5000 words

When 5000 words are used as input, the original code runs for about 1480 seconds (0-1480) and the optimized code runs for about 570 seconds (1480-2050) as shown in Figure 5.20.

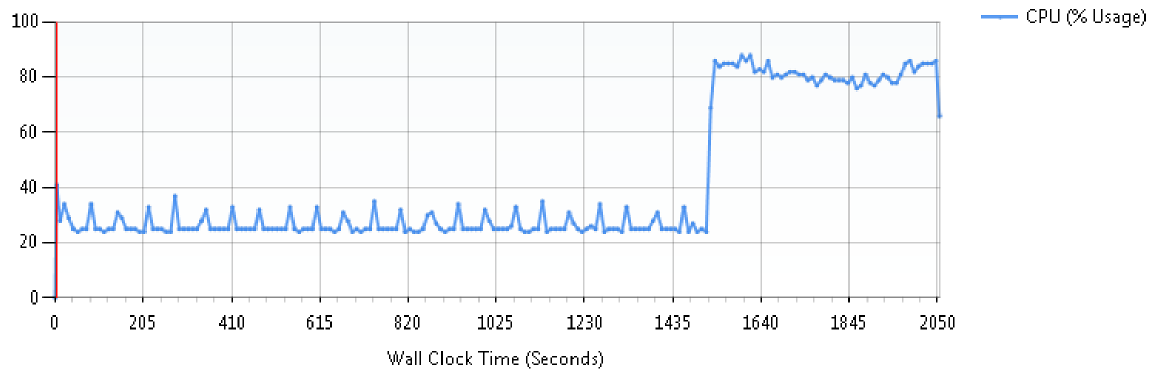


Figure 5.20 CPU usage of original and optimized code with 5000 words in 2 quad core machine

5.2.8 10000 words

As can be seen in Figure 5.21 when 10000 words are used as input, the original code runs for about 2990 seconds (0-2990) and the optimized code runs for about 1280 seconds (2990-4270).

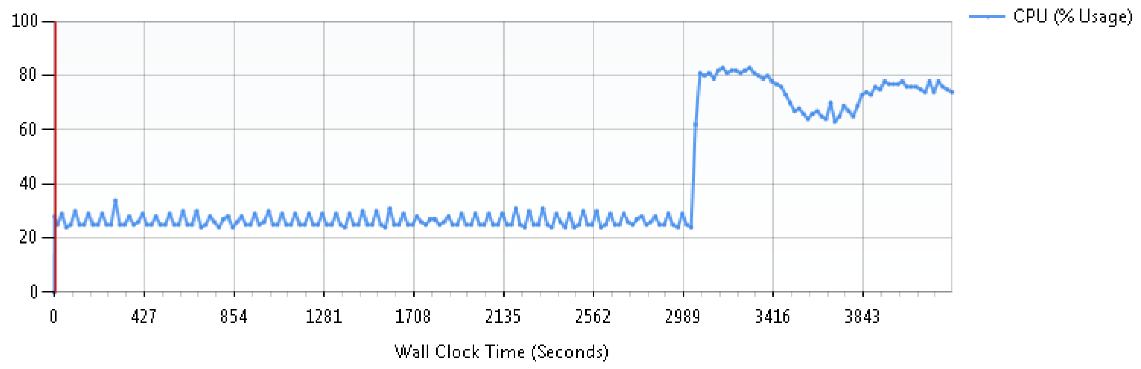


Figure 5.21 CPU usage of original and optimized code with 10000 words in 2 quad core machine

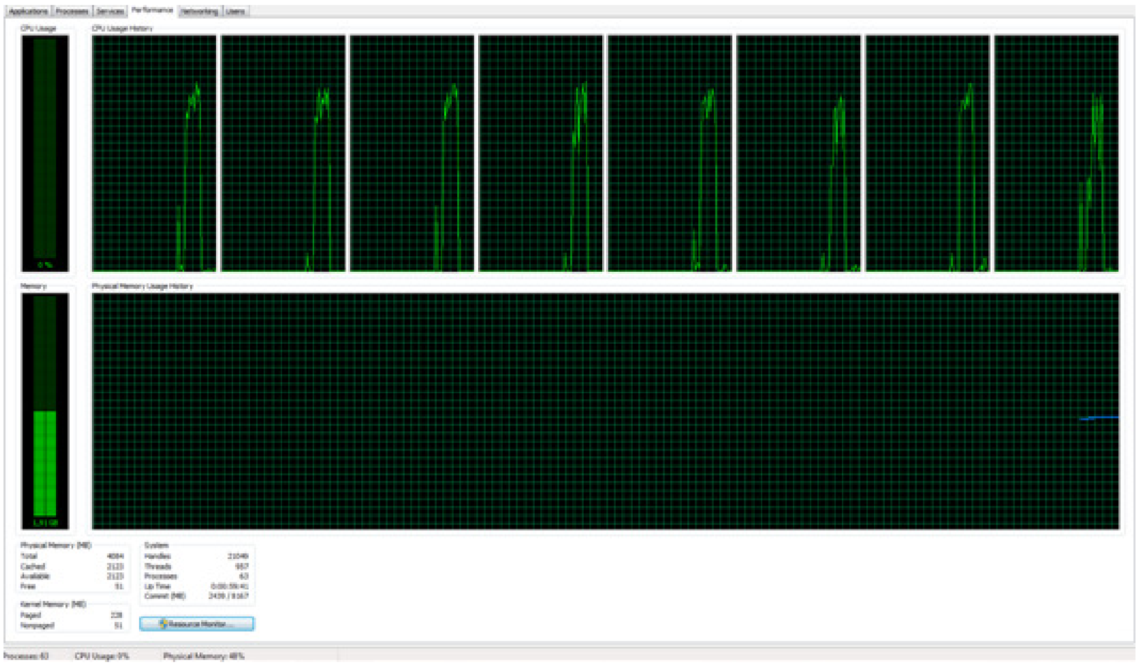
5.3 i7 - 8 Threaded Machine

This hardware design which can be seen in Figure 5.22 can be summarized as:

"Processor : Intel Core i7 CPU – 2.67 GHz"

"Memory: 4 GB RAM"

"System type: 64 bit Operating System"



Physical Memory (MB)
Total 4084
Cached 2123
Available 2123
Free 51

Kernel Memory (MB)
Paged 228
Nonpaged 51

System
Handles 21049
Threads 957
Processes 63
Up Time 0:00:59:41
Commit (MB) 2439 / 8167

Processes: 63 CPU Usage: 9% Physical Memory: 48%

Figure 5.22 Snapshot of processors in i7 8 threaded processor machine

5.3.1 50 words

When the input contains 50 words, the original code is runs for about 14 seconds (0-14) with 20 percent of the CPU and the optimized code runs for about 5 seconds (14-19) with about 80 percent of the CPU like shown in Figure 5.23.

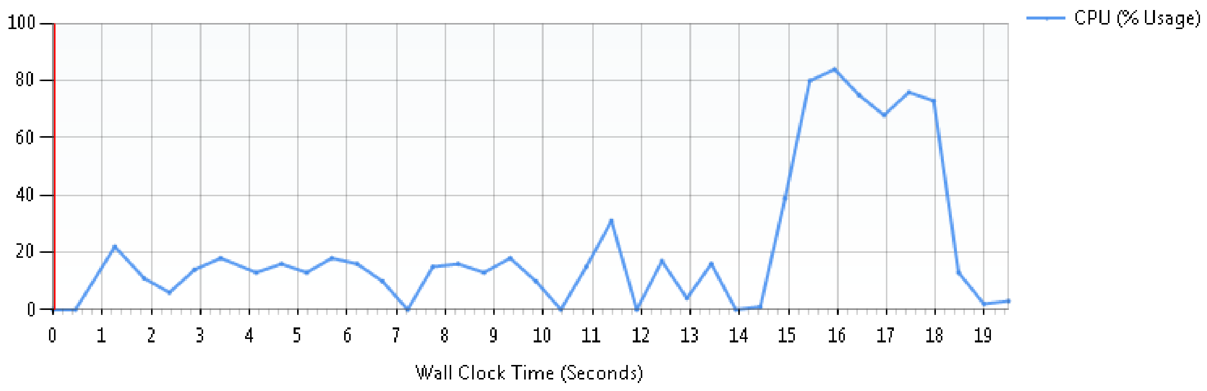


Figure 5.23 CPU usage of original and optimized code with 50 words in i7 8 threaded machine

5.3.2 100 words

As shown in Figure 5.24, when the input contains 100 words, the original code runs for about 26 seconds (0-26) with 20 percent of the CPU and the optimized code runs for about 8 seconds (26-34) with about 80 percent of the CPU.

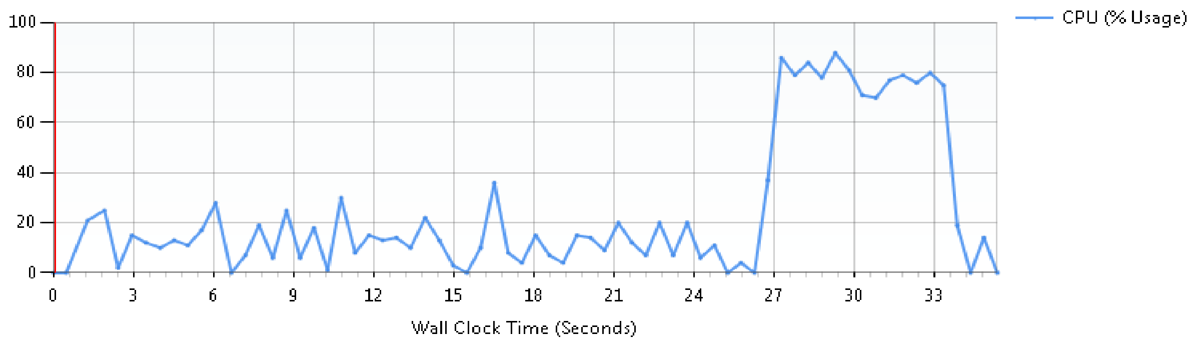


Figure 5.24 CPU usage of original and optimized code with 100 words in i7 8 threaded machine

5.3.3 150 words

As can be seen in Figure 5.25, when the input contains 150 words total execution time of the original code is about 37 seconds (0-36) with 20 percent usage of CPU. This is optimized to 13 seconds (36-49) with 80 percent usage of CPU.

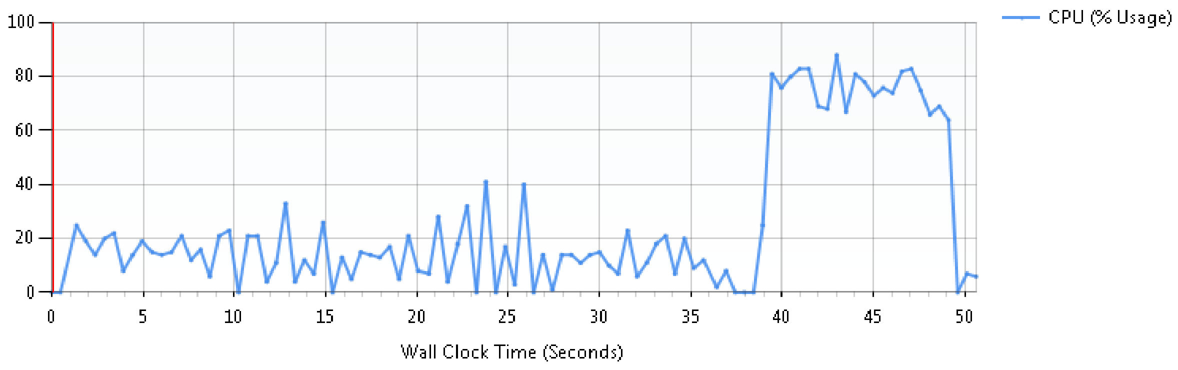


Figure 5.25 CPU usage of original and optimized code with 150 words in i7 8 threaded machine

5.3.4 300 words

With 300 words original code works for 72 seconds (0-72) and the optimized code works for 27 seconds (72-99). Figure 5.26 shows the graph of these results.

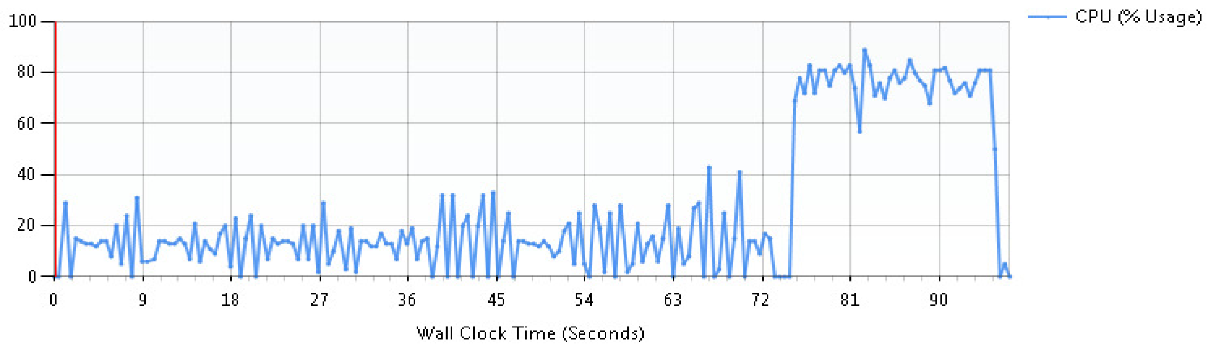


Figure 5.26 CPU usage of original and optimized code with 300 words in i7 8 threaded machine

5.3.5 600 words

With 600 words original code works for about 144 seconds (0-144) and the optimized code works for 45 seconds (144-189) as can be seen in Figure 5.27.

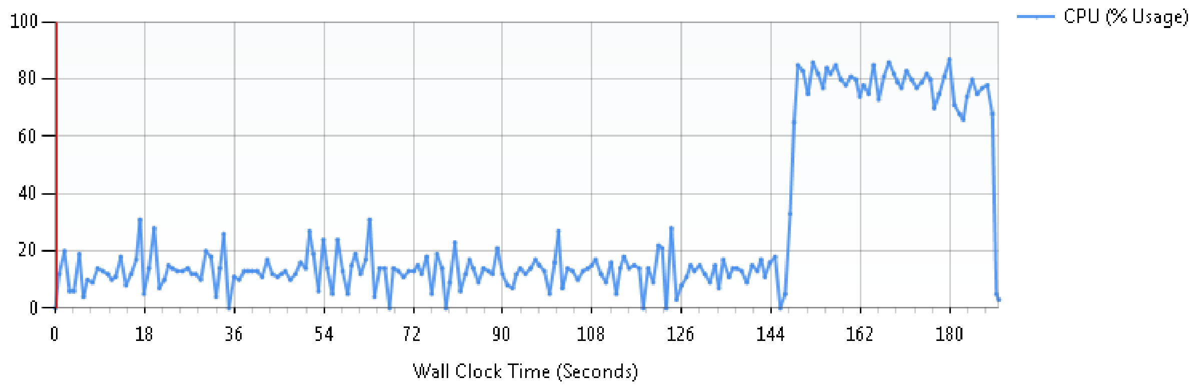


Figure 5.27 CPU usage of original and optimized code with 600 words in i7 8 threaded machine

5.3.6 1000 words

As shown in Figure 5.28, while working with 1000 words execution time is about 248 seconds (0-248) and execution time of the optimized code is about 62 seconds (248-310).

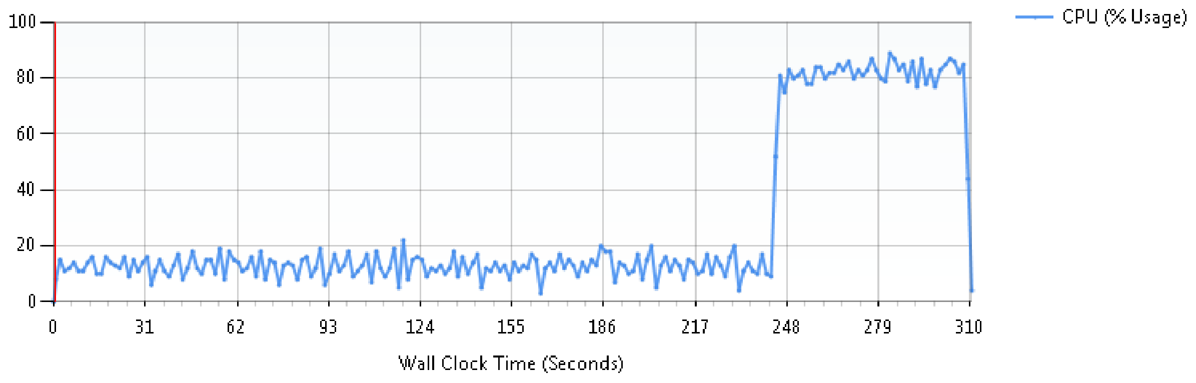


Figure 5.28 CPU usage of original and optimized code with 1000 words in i7 8 threaded machine

5.3.7 5000 words

It can be seen in Figure 5.29 that when the input size is increased to 5000 words the total execution time of the original code increases to 1224 seconds (0-1224) while the total execution time of the optimized code increases to only 306 seconds (1224-1530).

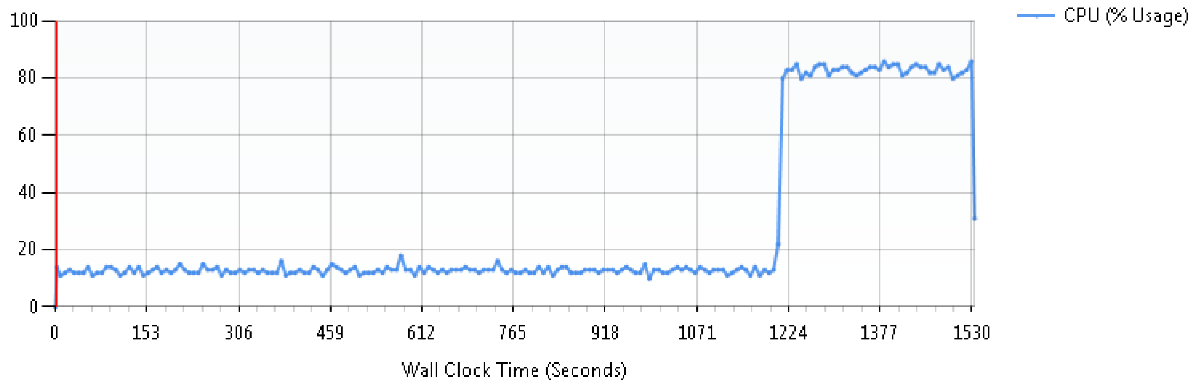


Figure 5.29 CPU usage of original and optimized code with 5000 words in i7 8 threaded machine

5.3.8 10000 words

When the input size is increased to 10000 words the difference between the total execution times of the original and the optimized code increases too. The original code works for about 2400 seconds (0-2400) (about 40 minutes) while the optimized code works for about 700 seconds (2400-3100) (about 10 minutes) as shown in Figure 5.30.

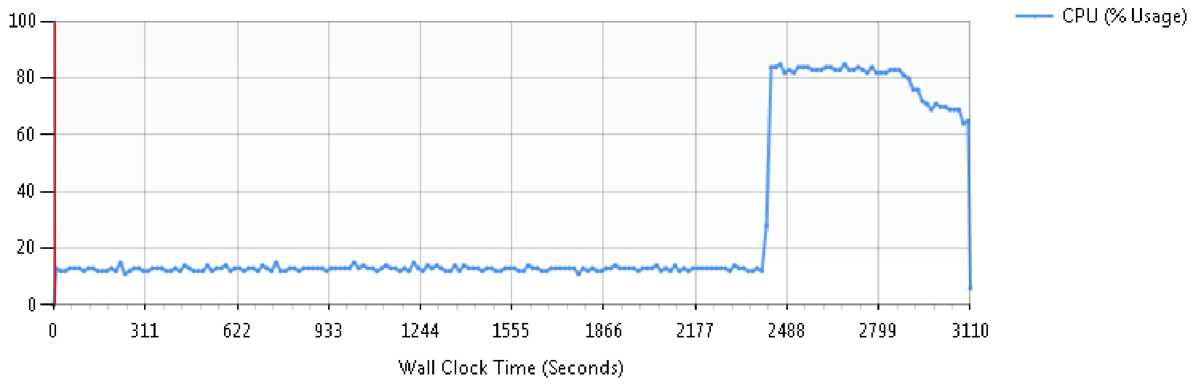


Figure 5.30 CPU usage of original and optimized code with 10000 words in i7 8 threaded machine

5.4 Results of tests

5.4.1 Performance results

As summary, the difference which occurs with the input size changes and hardware design difference can be seen in Figure 5.31, Figure 5.32 and Figure 5.33.

As you see in Figure 5.31, when the input contains 50 words on dual core machine, the difference between the total execution time of the original code and the optimized code is about 11 seconds and when the input data contains 10000 words, the difference increases to about 1220 seconds.

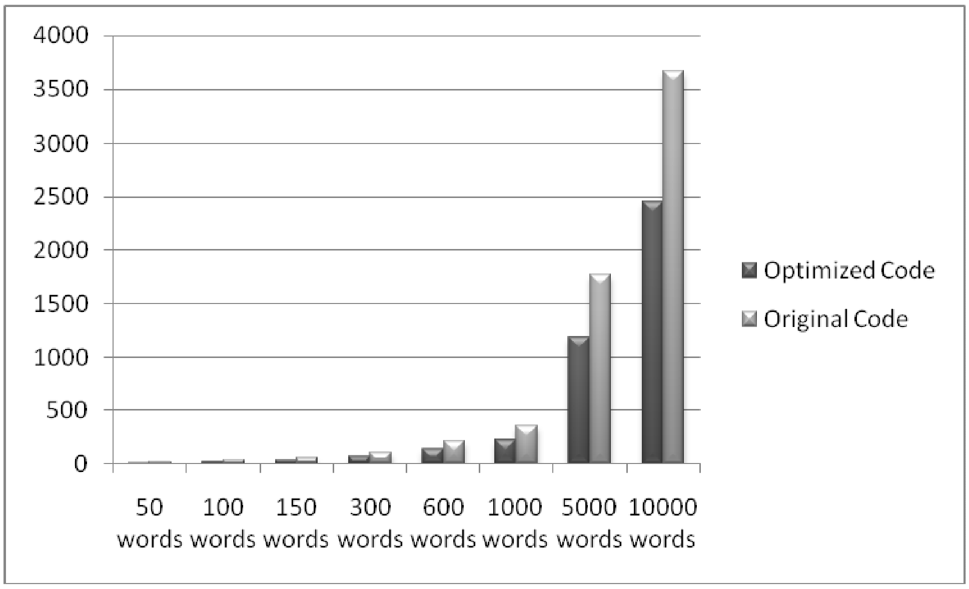


Figure 5.31 Summary of results of original and optimized code with different input sizes in dual core machine

As you see in Figure 5.32, when the input contains 50 words on quad core machine, the difference between the total execution time of the original code and the optimized code is about 10 seconds and when the input data contains 10000 words, the difference increases to about 1710 seconds.

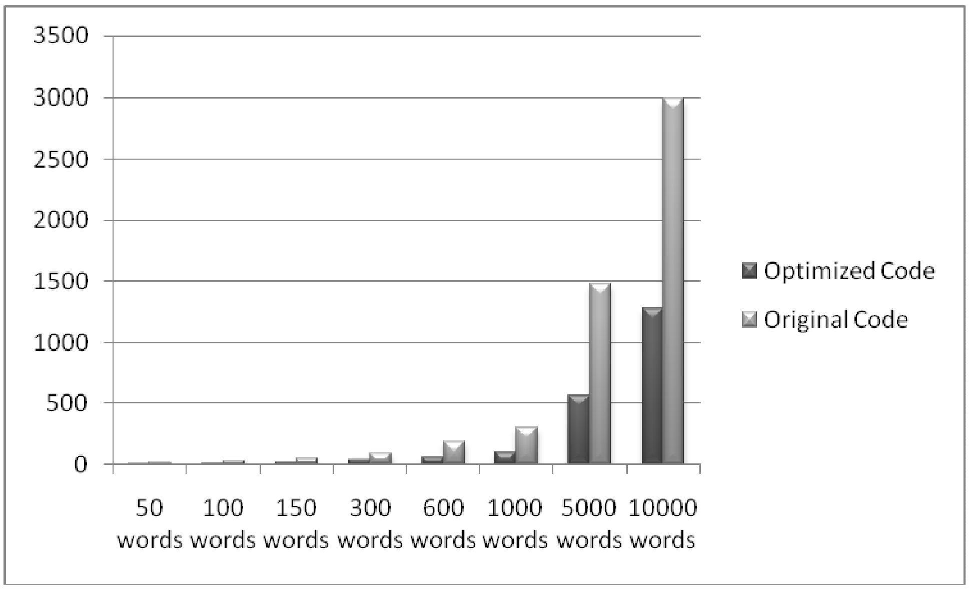


Figure 5.32 Summary of results of original and optimized code with different input sizes in 2 quad core machine

As you see in Figure 5.33, when the input contains 50 words on i7 8 threaded machine, the difference between the total execution time of the original code and the optimized code is about 9 seconds which can be ignored. But when the input data contains 10000 words, the difference increases to about 30 minutes which cannot be ignored.

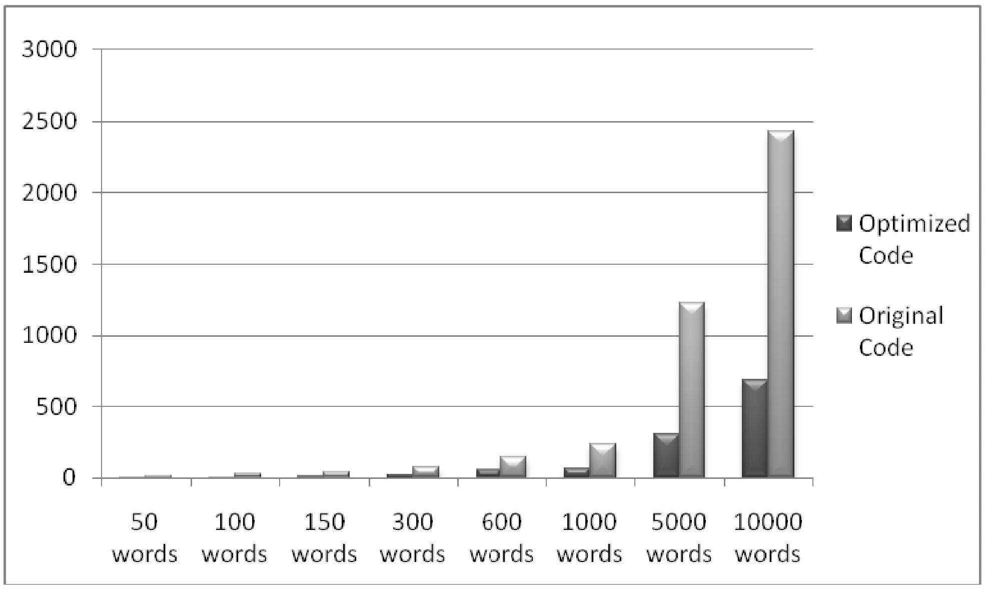


Figure 5.33 Summary of results of original and optimized code with different input sizes in i7 8 threaded machine

Figure 5.34 shows these results together where the effect of optimization can be seen clearly.

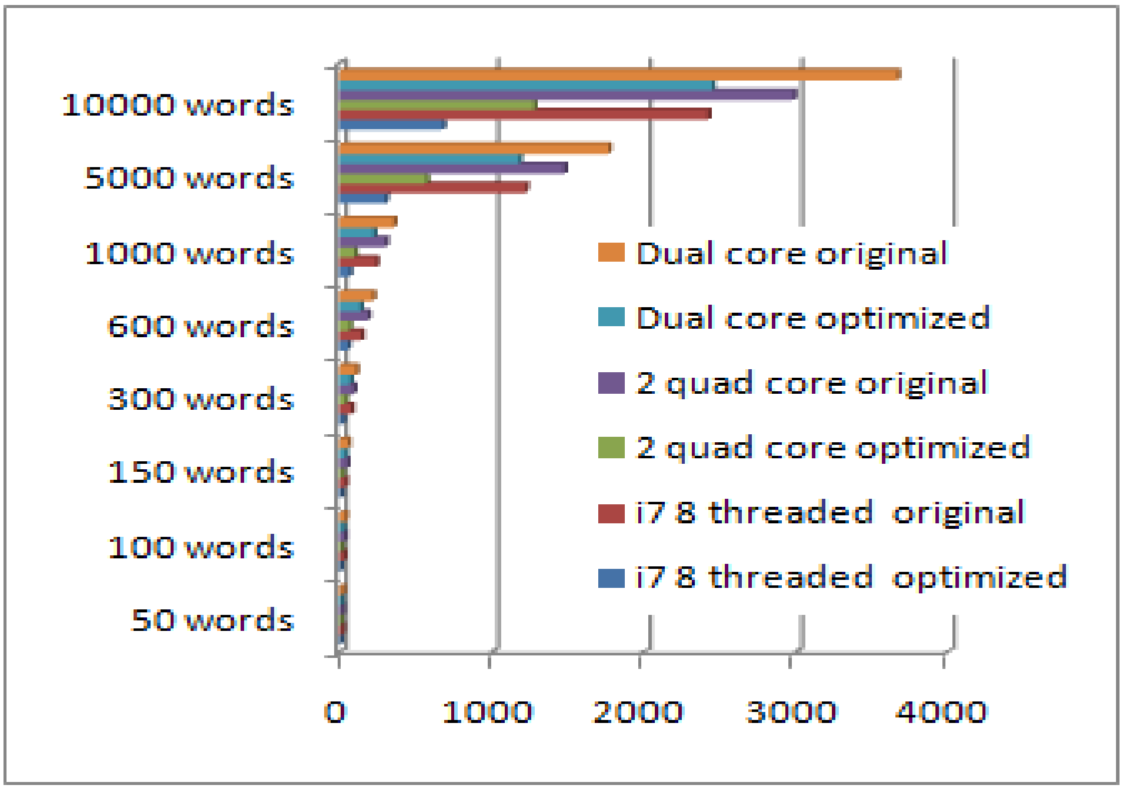


Figure 5.34 Summary of results of original and optimized code with different input sizes on different machine designs

5.4.2 Battery status results

To see the difference and effects in terms of battery usage, an application has been developed. In this application, the windows application analyzed in the previous section is executed between the battery power status checks of the machine. At first the notebook is fully charged. It is unplugged, and the application is started. It starts with checking the battery status of the machine. When it reaches to a starting point, the windows application is started with original source codes. When the application stops, the battery status is measured again and the result is saved. Then the machine is recharged fully and the same work is done for the application with optimized code.

Battery status change results according to different input sizes are given in Table 5.1 with their time costs in a dual core machine where bigger battery status change differences are expected in more threaded machines.

Table 5.1 Performance and battery usage comparison of original and optimized code with different input sizes in a dual core machine

	Original code		**Optimized code**	
	Execution time	Battery Status change	Execution time	Battery Status change
50 words	18 seconds	%99 - %99	11 seconds	%99 - %99
100 words	36 seconds	%99 - %99	24 seconds	%99 - %99
150 words	54 seconds	%99 - %99	30 seconds	%99 - %99
300 words	102 seconds	%99 - %98	63 seconds	%99 - %99
600 words	208 seconds	%99 - %95	169 seconds	%99 - %97
1000 words	343 seconds	%99 - %93	278 seconds	%99 - %95
5000 words	1723 seconds	%99 - %63	1057 seconds	%99 - %74
10000 words	3440 seconds	%99 - %30	2355 seconds	%99 - %43

By using the results shown in Table 5.1, curve fitting can be done to find an equation for a curve that fits this data. The data set can be defined as shown below:

X – independent variable (execution time difference of original and optimized code)

Y – dependent variable (battery status change difference of original and optimized code)

After performing curve fitting, an equation will be formed like "y = mx + C" where;

y = dependent variable, x = independent variable, m and C = constants

Here, curve fitting is applied on performance gain data (execution time difference of original and optimized code) and energy gain data (battery usage difference of original and optimized code) and result is shown in Figure 5.35.

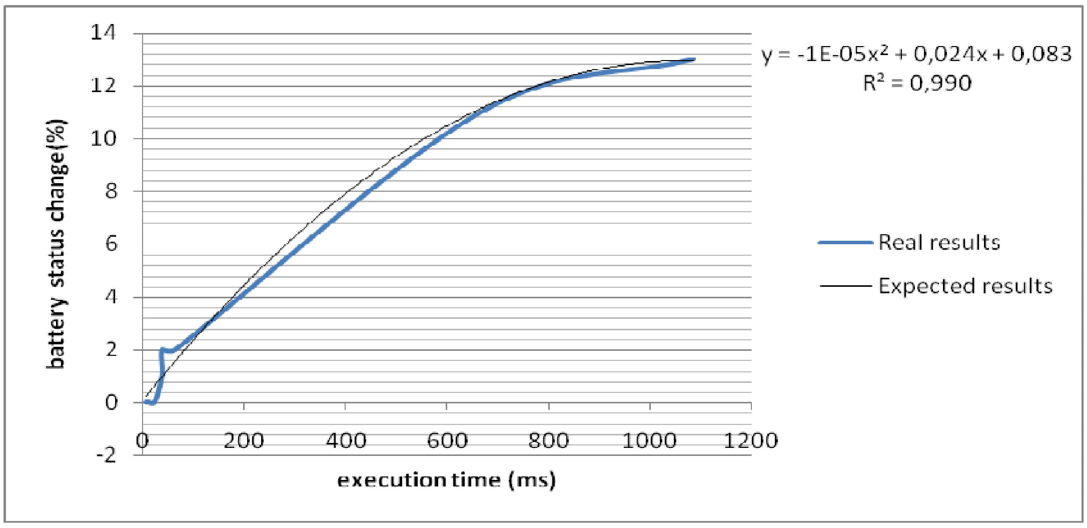


Figure 5.35 Curve fitting resuits for performance gain with battery life gain

5.4.3 Energy results

To see the difference and effects in terms of energy consumption, extra add-on has been used with the same application and with different input files. The device is a wattmeter that can measure the energy consumption quantity and calculate energy cost and it is shown in Figure 5.36.

Figure 5.36 Wattmeter

5.4.3.1 10000 words

The results of the energy consumption tests for 10000 words on different machines are given in Table 5.2.

Table 5.2 Energy consumption test results for 10000 words

		# of Threads	**wh**	**wh-Idle wh**	**Average watt**	**Real Time (min)**	**Real Time**
Test 1	**Optimized (Core i7)**	8	40	17,06	209,26	11,47	0:11:28
	Regular (Core i7)	8	100	20,64	151,21	39,68	0:39:41
Test 2	**Optimized (Core i7)**	8	40	17,35	211,89	11,33	0:11:20
	Regular (Core i7)	8	100	20,66	151,25	39,67	0:39:40
Test 3	**Optimized (Q6600)**	4	40	17,15	127,78	18,78	0:18:47
	Regular (Q6600)	4	87	26,23	104,69	49,67	0:49:40
Test 4	**Optimized (D525)**	2	10	1,28	9,18	65,37	1:05:22
	Regular (D525)	2	23	2,99	9,18	152,57	2:32:34
Test 5	**Optimized (Core2Duo)**	2	20	5,41	32,89	36,48	0:36:29
	Regular (Core2Duo)	2	30	7,37	31,82	56,56	0:56:34
Test 6	**Optimized (Core2Duo)**	2	20	6,09	34,50	34,78	0:34:47
	Regular (Core2Duo)	2	30	7,34	31,77	56,65	0:56:39
Test 7	**Optimized (Core2Duo)**	2	20	5,12	32,25	37,21	0:37:13
	Regular (Core2Duo)	2	30	7,36	31,80	56,61	0:56:37
Test 8	**Optimized (Core2Duo)**	2	20	4,53	31,02	38,68	0:38:41
	Regular (Core2Duo)	2	30	7,42	31,88	56,46	0:56:28
Test 9	**Optimized (Core2Duo)**	2	20	5,92	34,08	35,21	0:35:13
	Regular (Core2Duo)	2	30	7,40	31,86	56,49	0:56:29

5.4.3.2 50000 words

The results of the energy consumption tests for 50000 words on different machines are given in Table 5.3.

Table 5.3 Energy consumption test results for 50000 words

		# of Threads	**wh**	**wh-Idle wh**	**Average watt**	**Real Time (min)**	**Real Time**
Test 1	**Optimized (Core i7)**	8	223	85,42	194,32	68,96	1:08:57
	Regular (Core i7)	8	520	118,09	155,26	200,96	3:20:57
Test 2	**Optimized (Core i7)**	8	220	77,43	185,17	71,29	1:11:17
	Regular (Core i7)	8	520	117,09	154,87	201,46	3:21:27
Test 3	**Optimized (Core i7)**	8	230	91,98	199,97	69,01	1:09:01
	Regular (Core i7)	8	520	118,48	155,41	200,76	3:20:46
Test 4	**Optimized (Q6600)**	4	233	107,75	135,63	103,22	1:43:13
	Regular (Q6600)	4	460	151,50	108,85	253,56	4:13:34
Test 5	**Optimized (Q6600)**	4	230	105,82	135,20	102,07	1:42:04
	Regular (Q6600)	4	460	150,95	108,65	254,02	4:14:01
Test 6	**Optimized (Core2Duo)**	2	140	140,00	43,55	192,90	3:12:54
	Regular (Core2Duo)	2	180	180,00	37,25	289,95	4:49:57

As can be seen from the tables, tests have been done in different machines with different designs and by using different input files. The watt-hour (wh) is a unit of energy commonly used to measure electricity. One watt-hour is the amount of electrical energy equivalent to a one-watt load drawing power for one hour and here, "wh" column shows this quantity while executing the code. Also, the "wh-idle" column shows the amount of electrical energy for that machine uses when it is idle and does nothing. Watt is the unit of power and defined as one joule per second.

During the tests done at this step, the device shows the watt value and its average value for the executed code are given in “Average watt” column in the tables. It can also be calculated from the watt-hour values with the execution time given in the remaining columns of the table. For example; from the results of Test-1 in Table 5.2 by using the Equation 4 and Equation 5, it can be calculated:

W = 1 Joule/Second (4)

1 wh = 3600 Joule (5)

⇨ **209,26** = (**40***3600) / (**11,47***60)

By using the test results energy consumption of the original and optimized code can be calculated and compared. For example by using the test results of Test-1 shown in Table 5.2, cost of the original and optimized code can be calculated:

$\sum C_{optimized} = T_{optimized} * E_{optimized}$

= (68,96 min. / 60) * 223 wh

= 1,15 hours * 223 wh

= 256,45 wh

= **0,25 kwh**

$\sum C_{original} = T_{original} * E_{original}$

= (200,96 min. / 60) * 520 wh

= 3,35 hours * 520 wh

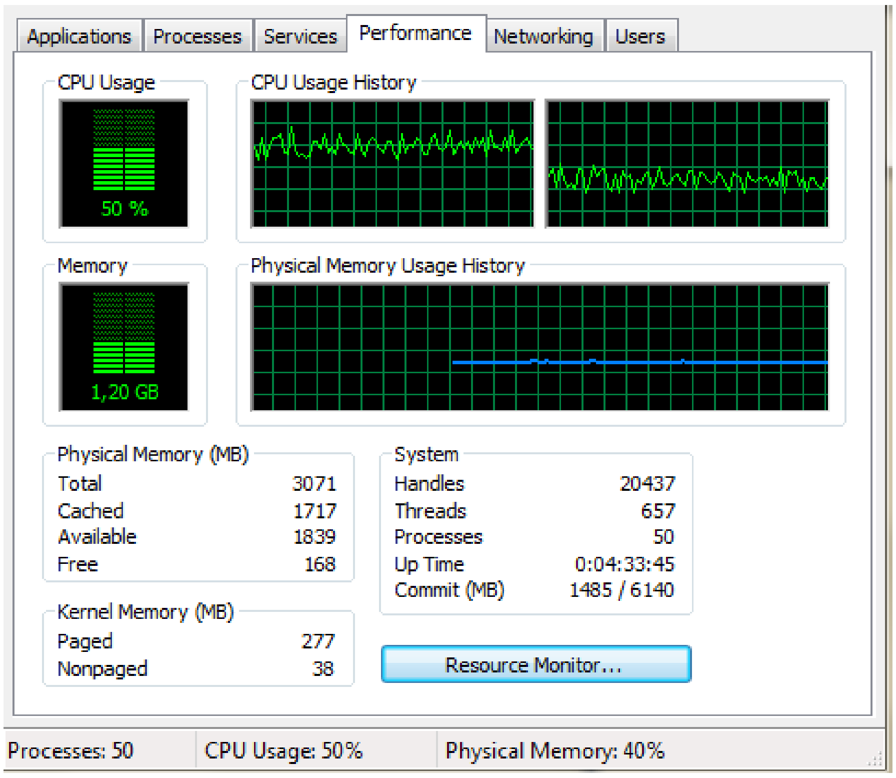
= 1742 wh

= **1,742 kwh**

Test results show that saving energy is possible by coding efficiently. Here, the effect of design of the machine and amount of the job done on energy consumption can be seen. Especially thread count is very effective on energy consumption that can be seen clearly from the results. As thread count increases the difference between original and optimized code gets bigger in terms of execution time and watt-hour value (energy). So, optimization done for less energy consumption results with bigger benefits.

5.4.4 Summary of results

As shown in previous sections, by using appropriate types, strategies and ways, execution time, battery usage and energy consumption can be decreased. During this decrement, memory and cpu usage of the applications can be worried. Figure 5.37 and Figure 5.38 show memory and cpu usage of the applications with original code and optimized code consecutively. As can be seen from these figures CPU usage increases in optimized code (because of parallel programming) and memory usage is almost same.



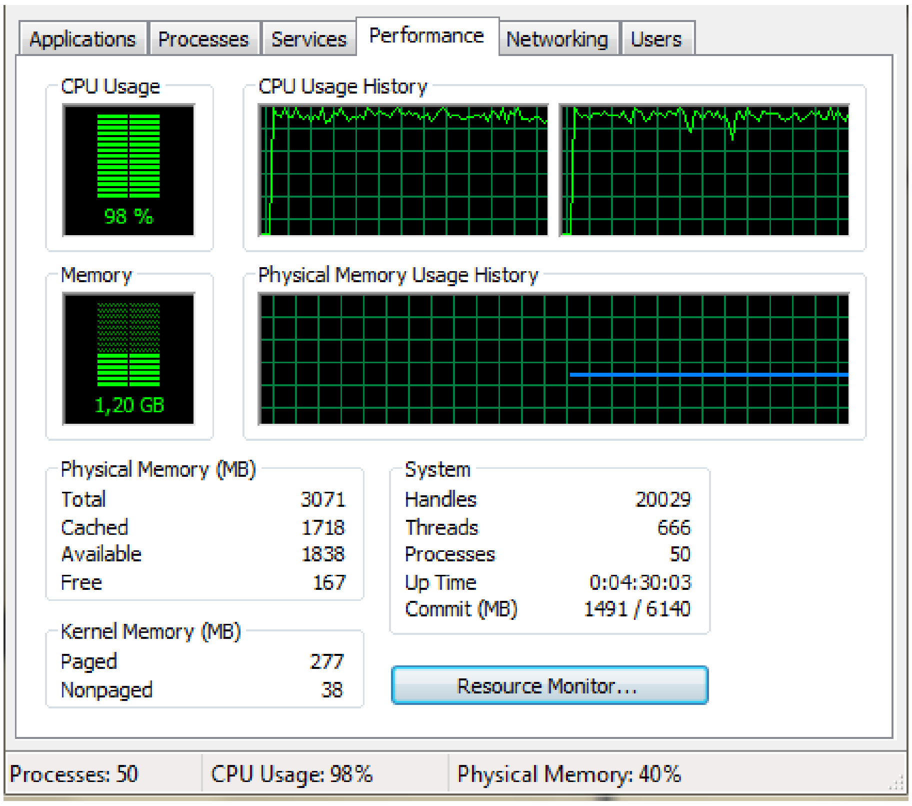
Physical Memory (MB)	
Total	3071
Cached	1717
Available	1839
Free	168

Kernel Memory (MB)	
Paged	277
Nonpaged	38

System	
Handles	20437
Threads	657
Processes	50
Up Time	0:04:33:45
Commit (MB)	1485 / 6140

Processes: 50 CPU Usage: 50% Physical Memory: 40%

Figure 5.37 Memory and cpu usage during execution of original code



Physical Memory (MB)

Total	3071
Cached	1718
Available	1838
Free	167

Kernel Memory (MB)

Paged	277
Nonpaged	38

System

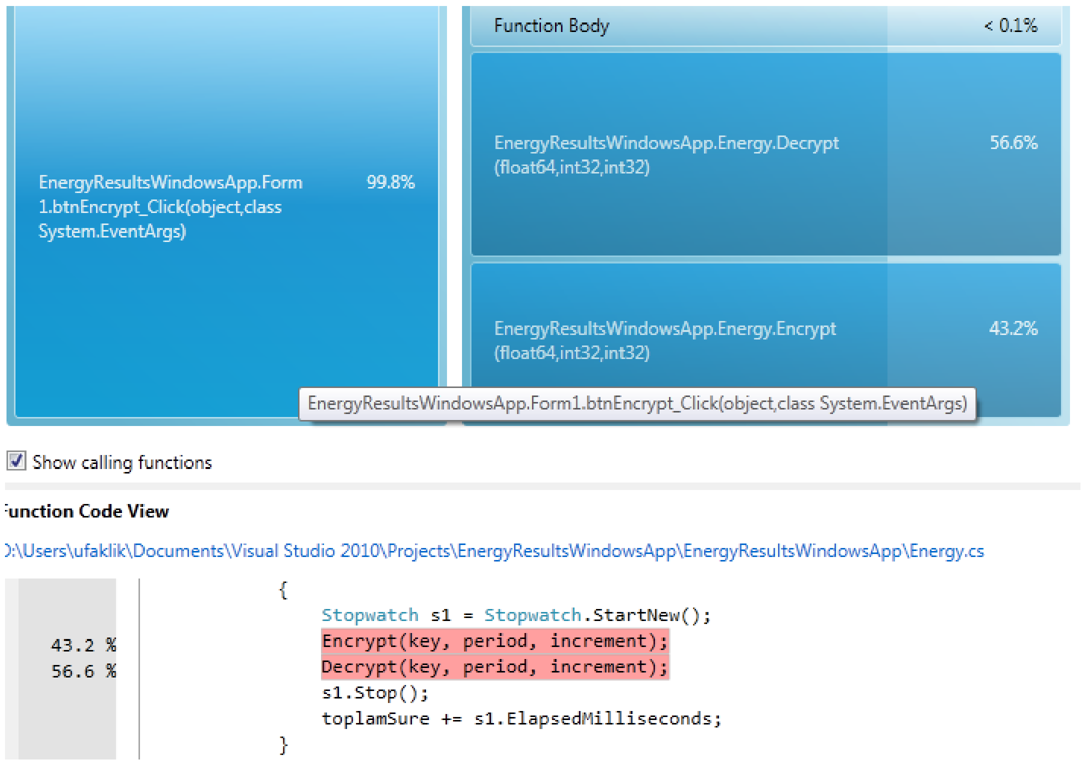
Handles	20029
Threads	666
Processes	50
Up Time	0:04:30:03
Commit (MB)	1491 / 6140

Processes: 50 CPU Usage: 98% Physical Memory: 40%

Figure 5.38 Memory and cpu usage during execution of optimized code

Also, it must be noted that all graphs that show cpu usage are not like a straight line and it is increasing and decreasing around average value. It can be because of a waiting of cpu for the disk or IO. Also different works done during the code cause different cpu utilization as shown in part of the report in Figure 5.39. As can be seen from the report, cpu usage of optimized code is about %99.8 but in details of code, cpu usage during decryption (%56.6) and encryption (%43.2) methods are also different, where there are differences in their details too.

Different tools have been used to see and compare the results on different environments. For example there is a functionality within the PowerCfg utility for evaluating system energy efficiency for the Windows family of operating systems which also enables system manufacturers to inspect a Windows platform for common energy efficiency problems. In Windows 7, the Windows PowerCfg utility is enhanced to detect many common energy efficiency problems, such as ineffective use of suspend by USB devices, excessive processor utilization, increased timer resolution, inefficient power policy settings, and battery capacity degradation.

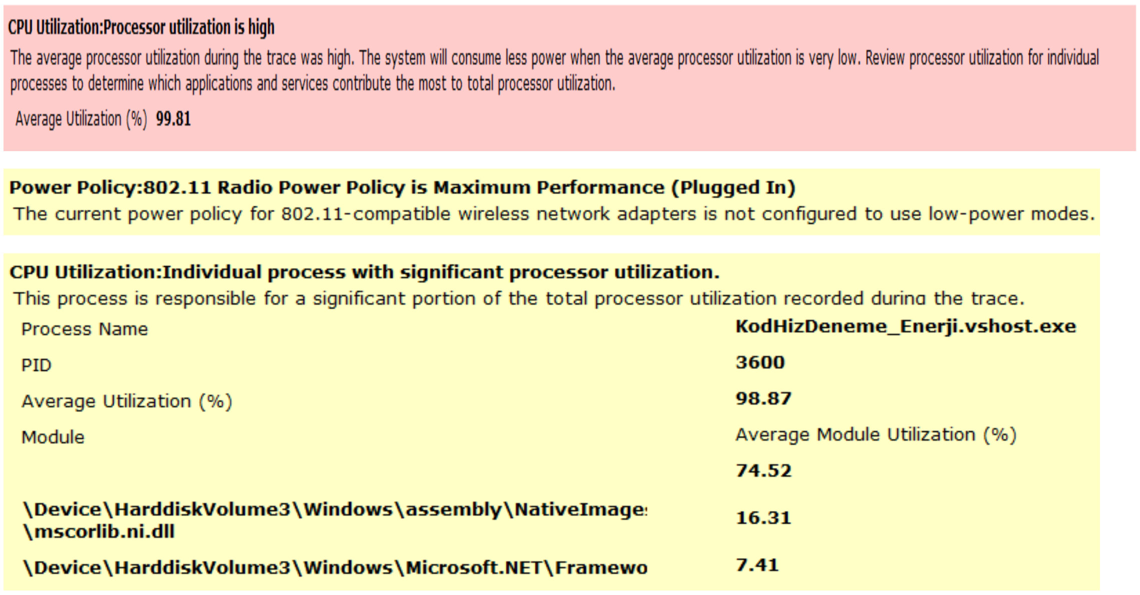

Function Body
< 0.1%
EnergyResultsWindowsApp.Form1.btnEncrypt_Click(object,class System.EventArgs)
99.8%
EnergyResultsWindowsApp.Energy.Decrypt (float64,int32,int32)
56.6%
EnergyResultsWindowsApp.Energy.Encrypt (float64,int32,int32)
43.2%
EnergyResultsWindowsApp.Form1.btnEncrypt_Click(object,class System.EventArgs)
Show calling functions
unction Code View
:\Users\ufaklik\Documents\Visual Studio 2010\Projects\EnergyResultsWindowsApp\EnergyResultsWindowsApp\Energy.cs

```
             {
                 Stopwatch s1 = Stopwatch.StartNew();
 43.2 %          Encrypt(key, period, increment);
 56.6 %          Decrypt(key, period, increment);
                 s1.Stop();
                 toplamSure += s1.ElapsedMilliseconds;
             }
```

Figure 5.39 CPU usage details of optimized code

It also generates an HTML-formatted report that contains details about each problem that it detected. Here, this analysis has been done for original and optimized code separately and two reports have been generated. After getting these reports the results reached in previous sections are supported. All environmental parameters are same which shows that the tests are done under the same conditions for original and optimized code. Also cpu utilization of applications are same with the previous results as shown in Figure 5.40 and Figure 5.41.

As a result, two different applications which do the same work and give the same output did their work in different times with different battery usage and this difference gets bigger as the job being done gets bigger. Think that you are out where you cannot charge your notebook having low battery power on which you have to do a job. In this case, if you use optimized code you can see the work results before it becomes empty where maybe the battery power will be insufficient for the original one.



CPU Utilization:Processor utilization is high

The average processor utilization during the trace was high. The system will consume less power when the average processor utilization is very low. Review processor utilization for individual processes to determine which applications and services contribute the most to total processor utilization.

Average Utilization (%) **99.81**

Power Policy:802.11 Radio Power Policy is Maximum Performance (Plugged In)

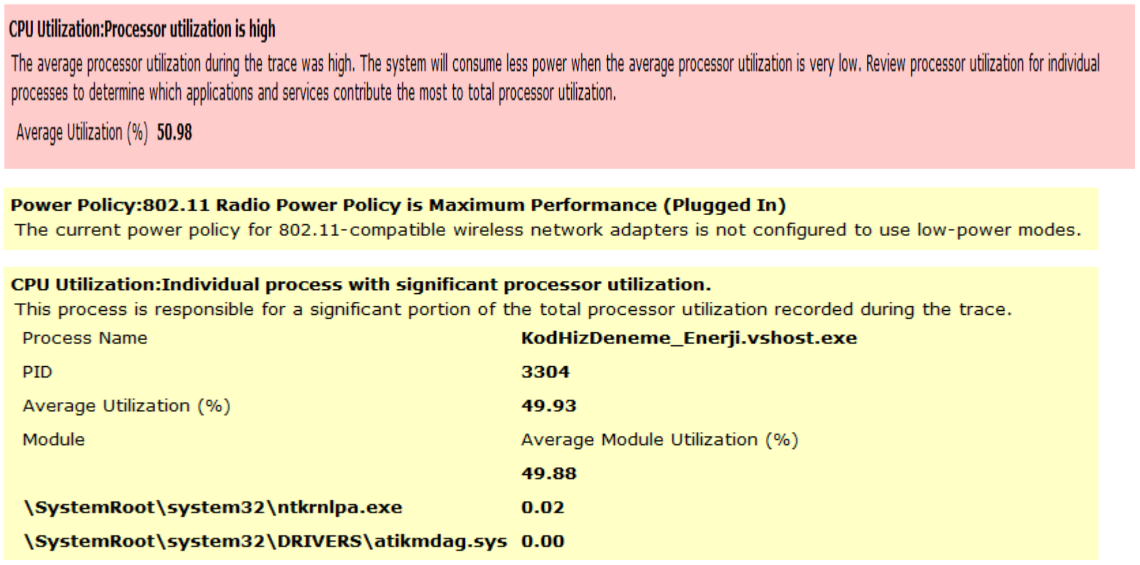
The current power policy for 802.11-compatible wireless network adapters is not configured to use low-power modes.

CPU Utilization:Individual process with significant processor utilization.

This process is responsible for a significant portion of the total processor utilization recorded during the trace.

Process Name	**KodHizDeneme_Enerji.vshost.exe**
PID	**3600**
Average Utilization (%)	**98.87**
Module	Average Module Utilization (%)
	74.52
\Device\HarddiskVolume3\Windows\assembly\NativeImage: \mscorlib.ni.dll	**16.31**
\Device\HarddiskVolume3\Windows\Microsoft.NET\Framewo	**7.41**

Figure 5.40 A part of report generated with PowerCfg utility while executing optimized code



CPU Utilization:Processor utilization is high

The average processor utilization during the trace was high. The system will consume less power when the average processor utilization is very low. Review processor utilization for individual processes to determine which applications and services contribute the most to total processor utilization.

Average Utilization (%) **50.98**

Power Policy:802.11 Radio Power Policy is Maximum Performance (Plugged In)

The current power policy for 802.11-compatible wireless network adapters is not configured to use low-power modes.

CPU Utilization:Individual process with significant processor utilization.

This process is responsible for a significant portion of the total processor utilization recorded during the trace.

Process Name	**KodHizDeneme_Enerji.vshost.exe**
PID	**3304**
Average Utilization (%)	**49.93**
Module	Average Module Utilization (%)
	49.88
\SystemRoot\system32\ntkrnlpa.exe	**0.02**
\SystemRoot\system32\DRIVERS\atikmdag.sys	**0.00**

Figure 5.41 A part of report generated with PowerCfg utility while executing original code

CHAPTER SIX

CONCLUSIONS

With the increasing role of power-conscious systems in our lives, energy consumption gained more importance and by the way, as battery systems cannot long last, power usage is still a major concern studied from the perspective of software.

In this thesis, the aim is finding various ways, types and techniques at the software implementation level (especially within OOP development) that use lower energy while providing same output.

Results show that, the time CPU spends is parallel with the battery usage and energy consumption, so it is the basic approach in this research that ‘if the application works faster it consumes less energy with the help of increased idle time’. In addition to previous works done before, contributions of this thesis are;

- ⇨ High performance can be supplied by optimizing the source code.
- ⇨ Battery usage of the code is generally parallel with the execution time of it and long battery life can be supplied by using appropriate strategy or type in the source code.
- ⇨ Energy consumption can generally be estimated and writing energy efficient code is possible.
- ⇨ Execution time of the programs gives clues about energy consumption because longer execution time generally means less idle time and more energy consumption.
- ⇨ Optimizing software from the higher levels possible, it is cheaper and easier.
- ⇨ Design of the hardware (also the number of threads) and amount of the job done affect the performance and energy consumption of the software.

It would be very useful if there is a tool or algorithm that converts an existing code to its lower-power counterpart. So finally, this study can be used in developing automatic techniques for determining the energy consumption of applications and decreasing the energy consumption with software optimization techniques.

REFERENCES

Allen, S. (2010). *C# Method Parameter Performance and Registers.* Retrieved February 15, 2010, from http://dotnetperls.com/method-parameter

Applications Power Management. (n.d.). Retrieved March 07, 2010, from http://lesswatts.org/projects/applications-power-management/

Aslan, K. (2006). *Derleyicilerin kod optimizasyonu.* Retrieved May 10, 2010, from http://www.kaanaslan.com/resource/article/display_article.php?page=1&id=75

Best Practices For Writing High Performance Code. (2010). Retrieved May 10, 2010, from http://www.csharphelp.com/2010/02/c-best-practices-to-write-high-performance-code/

Boxing and Unboxing. (2010). Retrieved February 15, 2010, from http://msdn microsoft.com/en-us/library/yz2be5wk.aspx

Chantarasathaporn, K., & Srisa-an, C. (2006). *Object-Oriented Programming Strategies in C# for Power Conscious System.* Retrieved January 5, 2010, from http://www.waset.org/journals/waset/v10/v10-17.pdf

Chatzigeorgiou, A. (2002). *Performance and power evaluation of C++ object-oriented programming in embedded processors.* Retrieved November 04, 2010, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.6227&rep=rep 1&type=pdf

Flinn, J., & Satyanarayanan, M. (1999). *Energy-aware adaptation for mobile applications.* Retrieved June 8, 2010, from http://portal.acm.org/citation.cfm?id= 319155

Gusev, A. (2006). *Simple rules that boost your mobile application's performance.* Retrieved October 13, 2009, from http://www.developer.com/ws/data/article

.php/10944_3643266_1/Simple-Rules-that-Boost-Your-Mobile-Applications-Performance.htm

Kumar, V., & Gupta, A. (1993). *Analyzing Scalability of Parallel Algorithms and Architectures.* Retrieved May 21, 2010, from http://portal.acm.org/citation.cfm?id =186528.186531

Larsson, P. (2008). *Energy-Efficient Software Guidelines.* Retrieved May 12, 2010, from http://software.intel.com/en-us/articles/energy-efficient-software-guidelines/

Lee, M., & Tiwari, V., & Malik, S., & Fujita, M. (1995). *Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software.* Retrieved January 10, 2010, from http://portal.acm.org/citation.cfm?id=224486.224525

Leijen, D., & Hall, J. (2007). *Optimized Managed Code For Multi-Core Machines.* Retrieved May 16, 2010, from http://msdn.microsoft.com/en-us/magazine/cc 163340.aspx

LessWatts – Saving Power With Linux. (n.d.). Retrieved February 04, 2010, from http://www.lesswatts.org/projects/applications-power-management/avoid-pulling php

Miettinen, A. P., & Hirvisalo, V. (2009). *Energy-efficient parallel software for mobile hand-held devices.* Retrieved May 24, 2010, from http://portal.acm.org/ citation.cfm?id=1855591.1855603

Naik, K., & Wei, S.L. (2001). *Software Implementation Strategies for Power-Conscious Systems.* Retrieved January 5, 2010, from http://portal.acm.org/citation cfm?id=383768

Rodriguez, J. & Dutta, S. (2008). *Writing High Performance .NET Code.* Retrieved December 18, 2009, from http://software.intel.com/en-us/articles/writing-high-performance-net-code/

Scarpazza, D. (2006). *A source-level estimation and optimization methodology for execution time and energy consumption of embedded software.* Retrieved February 7, 2010, from http://citeseerx.ist.psu.edu/viewdoc/download?doi= 10.1.1.127.3267&rep=rep1&type=pdf

Simunic, T. & Benini, L. & Micheli, G. & Hans, M. (1999). *Source Code Optimization and Profiling of Energy Consumption in Embedded Systems.* Retrieved January 8, 2010, from http://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/2000/isss00/pdffiles/10_3.pdf

Steigerwald, B., & Chabukswar, R., & Krishnan, K., & Vega, J.D. (2007). *Creating Energy - Efficient Software.* Retrieved February 5, 2010, from http://software. intel.com/en-us/articles/creating-energy-efficient-software-part-1/

Steinke, S., & Schwarz, L., & Wehmeyer, L., & Marwedel, P. (2001). *Low Power Code Generation for a RISC Processor by Register Pipelining.*

Stemen, P. (2008). *Extending battery life with energy efficient applications.* Retrieved June 21, 2009, from http://channel9.msdn.com/pdc2008/PC02/

Tiwari, V., & Malik, S., & Wolfe, A. (1994). *Compilation Techniques for Low Energy: An Overview.* Retrieved June 21, 2009, from http://ieeexplore. ieee.org/xpl/freeabs_all.jsp?arnumber=573195

Tiwari, V., & Malik, S., & Wolfe, A. (1996). *Instruction Level Power Analysis and Optimization of Software.* Retrieved June 21, 2009, from http://www.springerlink .com/content/v0n5573147686547/

Toub, S. (2010). *Patterns of parallel programming.* Retrieved March 18, 2010, from http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en

Varszegi, J. (2004). Retrieved May 7, 2010, from http://dotnet.sys-con.com/node/46342

Yağmur, O. (2004). *Kod Optimizasyonu ve Taşınabilir Programlar Üretmek. Retrieved January 6, 2010, from* http://www.csharpnedir.com/articles/read/?filter=&author=&cat=c&id=296&title=Kod%20Optimizasyonu%20ve%20Ta%C5%9F%C4%B1nabilir%20Programlar%20%C3%9Cretmek

Şenyurt, B.S. (2010). *Paralel Programlamada Performans, Hız, Verimlilik ve Ölçeklenebilirlik Ölçümleri.* Retrieved June 20, 2010, from http://www.buraksenyurt.com/category/Parallel-Programming.aspx

APPENDICES

A sample code used in comparing styles and types in the specified language is below. It is the example of comparing recursion and iteration.

```
 private int TestRecursive(int p1)
 {
     if (p1 <= 1) return p1;
     int result = p1 + TestRecursive(p1 - 1);
     return result;
 }

 private int TestNonRecursive(int p1)
 {
     int result = 0;
     while (p1 > 0)  {
         result = result + p1;
         p1--;
     }
     return result;
 }

 private void Compare()
 {
     Stopwatch s1 = Stopwatch.StartNew();
     int res = 0;
     for (int i = 1; i < 10000; i++)
     {
         res += TestRecursive(i);
     }
     s1.Stop();

     res = 0;
     Stopwatch s2 = Stopwatch.StartNew();
     for (int i = 1; i < 10000; i++)
     {
         res += TestNonRecursive(i);
     }
     s2.Stop();
}
```

The sample code used in comparing battery usage is shown below (A timer is used and in every tick it checks the battery status for reaching a starting point, if it reaches, the test code is started and when it stops the battery status is measured again to see the energy consumption of the code):

```
private void timer1_Tick(object sender, EventArgs e)
{
  if (!_hasStarted)
  {
     if (GetBatteryStatus() == "99")
     {

        lblStart.Text += DateTime.Now.ToString() + "("
        + GetBatteryStatus() + ")";
        _hasStarted = true;
        _p =
        Process.Start(@"EnergyResultsWindowsApp.exe");
     }
  }
  else {
    if (!IsProcessOpen("EnergyResultsWindowsApp")) {
         timer1.Enabled = false;
         lblEnd.Text += DateTime.Now.ToString() + "(" +
        GetBatteryStatus() + ")";
    }
  }

}
```